Loon

Interactive graphics in R

R.W. Oldford

UNIVERSITY OF WATERLOO

The loon package

Loon is an interactive visualization system built using tcltk.

The loon package is available on CRAN. To install the package start your R and run

```
install.packages('loon')
```

You can also install the latest development release (make sure it looks like it is passing the "builds") directly from GitHub with the following R code (you might need to install devtools)

```
devtools::install_github("great-northern-diver/loon", subdir = "R")
```

Once installed, the loon package is loaded in the usual way:

```
library(loon)
```

And instructions on loon are available in different ways:

```
l_help()                    # loon's web overview
l_web()                     # loon's web manual
help(package = "loon")      # loon's R help pages
```

l_plot()

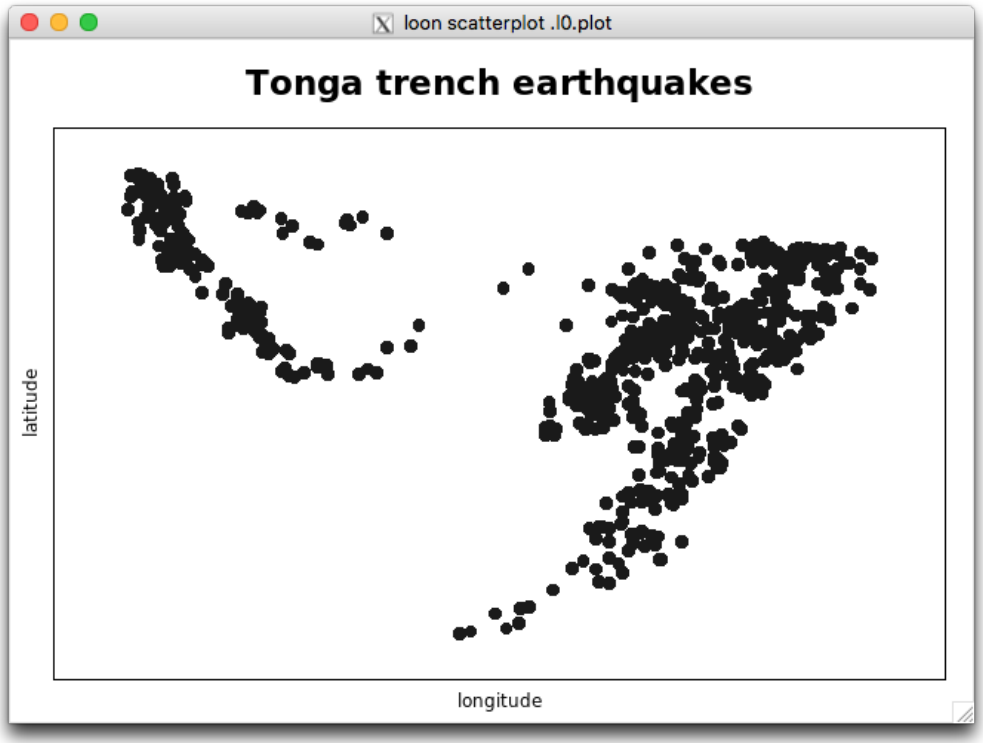
The basic plot function in `loon` is similar to that of `plot()` from the base `graphics` package.

```
# Tonga Trench Earthquakes
p <- l_plot(quakes$long, quakes$lat, title = "Tonga trench earthquakes",
            xlabel = "longitude", ylabel ="latitude")
# produces the scatterplot
```

l_plot()

The basic plot function in `loon` is similar to that of `plot()` from the base `graphics` package.

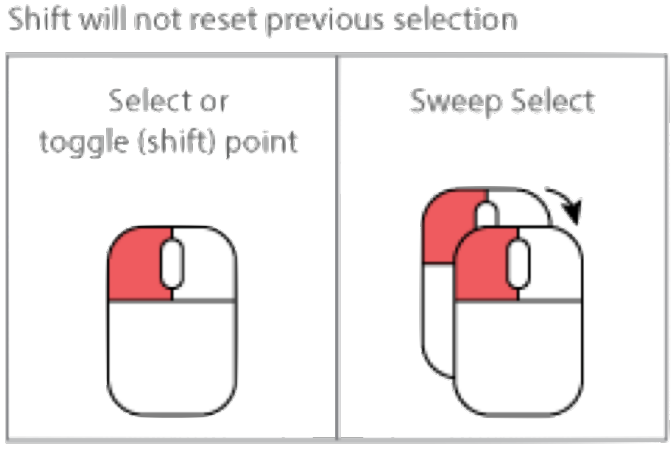
```
# Tonga Trench Earthquakes
p <- l_plot(quakes$long, quakes$lat, title = "Tonga trench earthquakes",
            xlabel = "longitude", ylabel ="latitude")
# produces the scatterplot
```



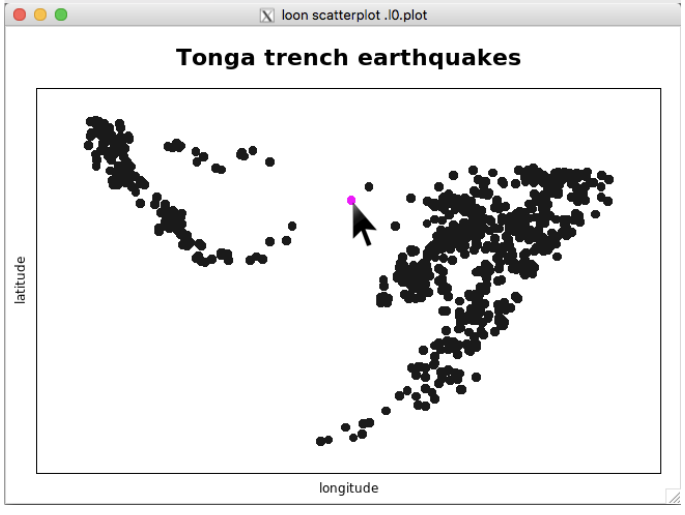
Interact with the plot using the mouse and keyboard.

Mouse gestures - selection

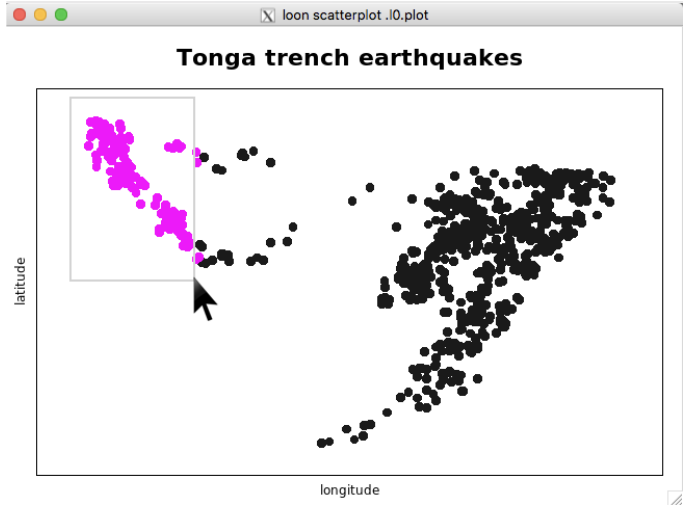
Points can be selected



One at a time

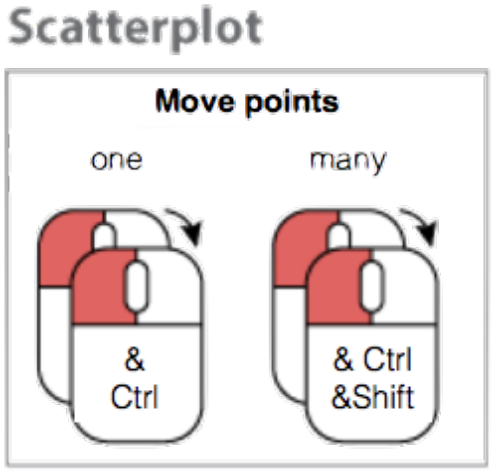


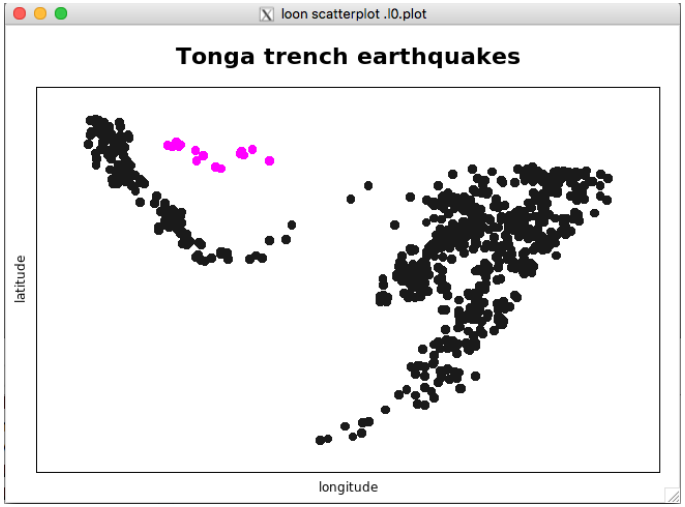
or by "sweeping" out a rectangle

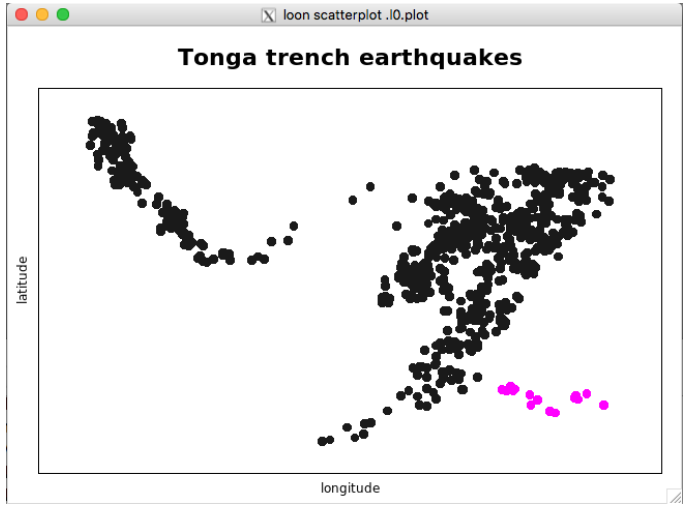


Mouse gestures - moving points

Selected points can be moved

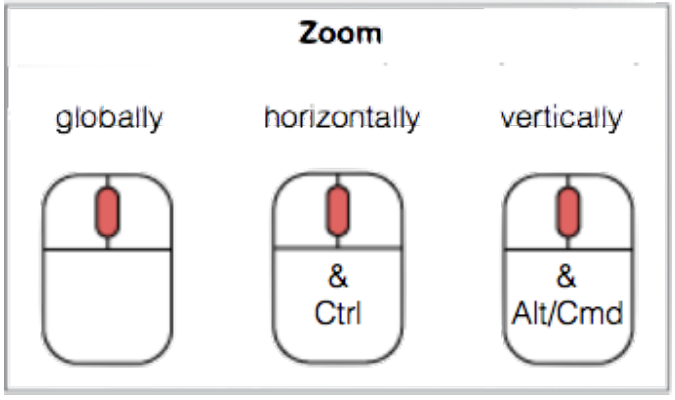




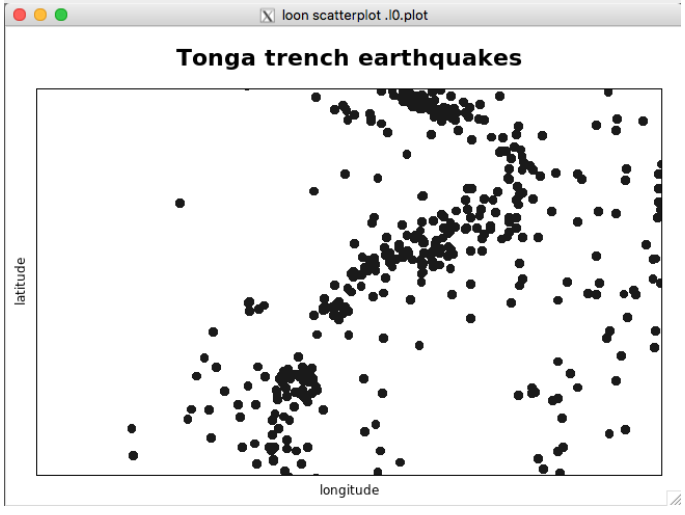


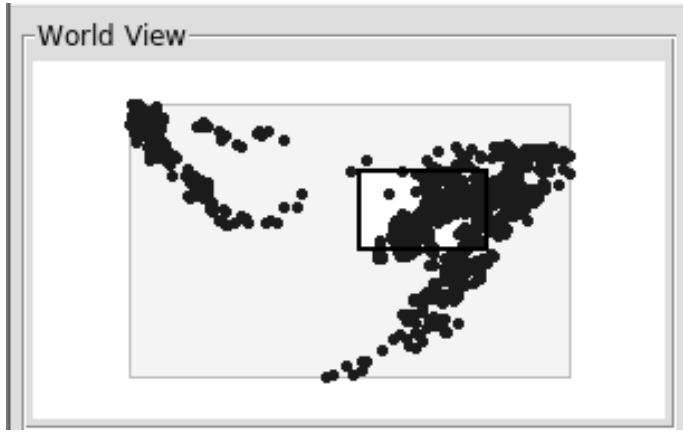
Mouse gestures - zooming

Zooming (on plot OR on "World View")



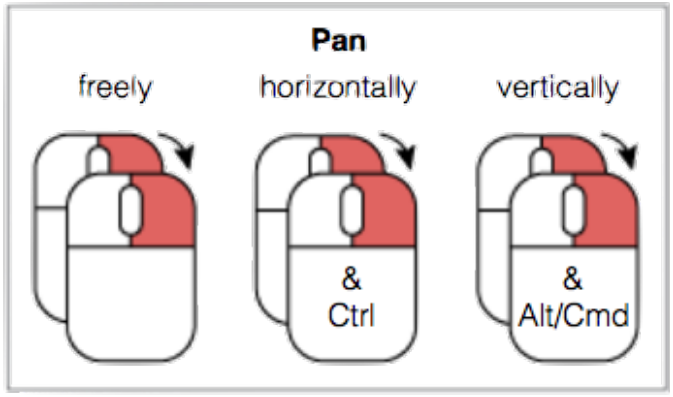
Zooming (globally)



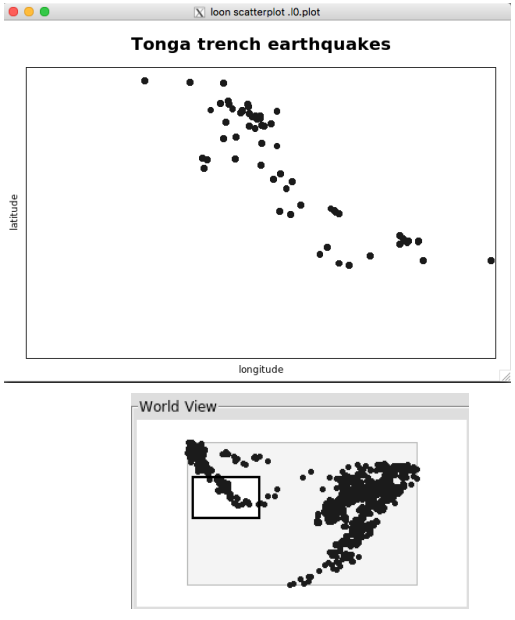


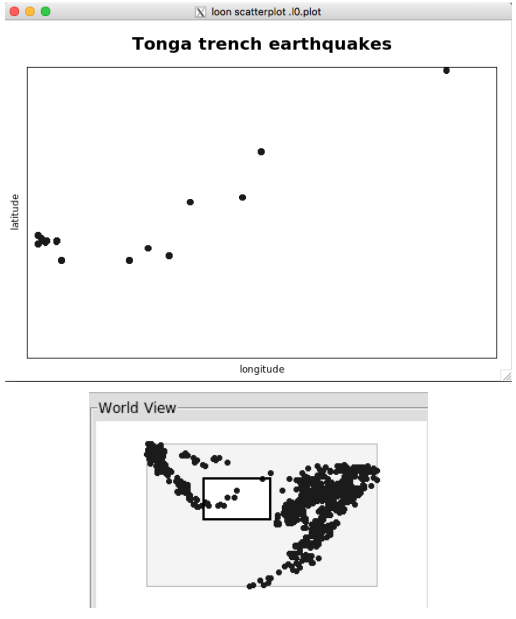
Mouse gestures - panning

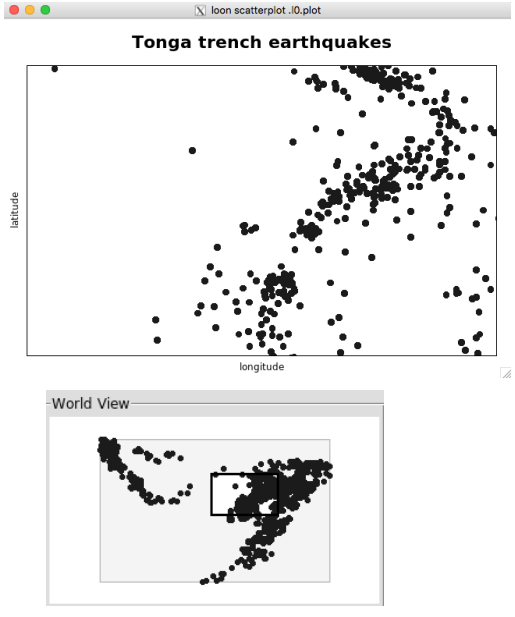
Panning (on plot OR on "World View")



Panning (horizontally)

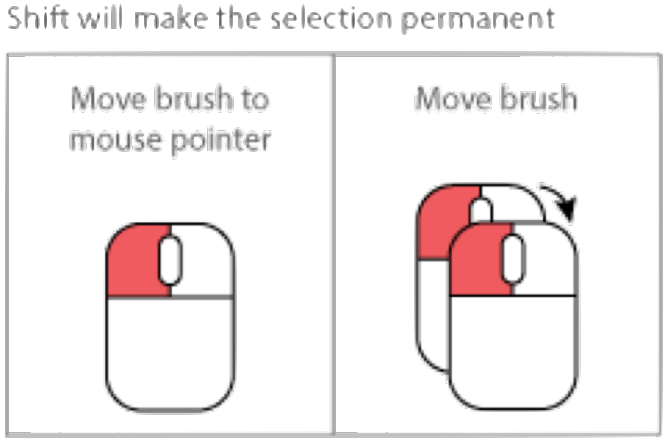


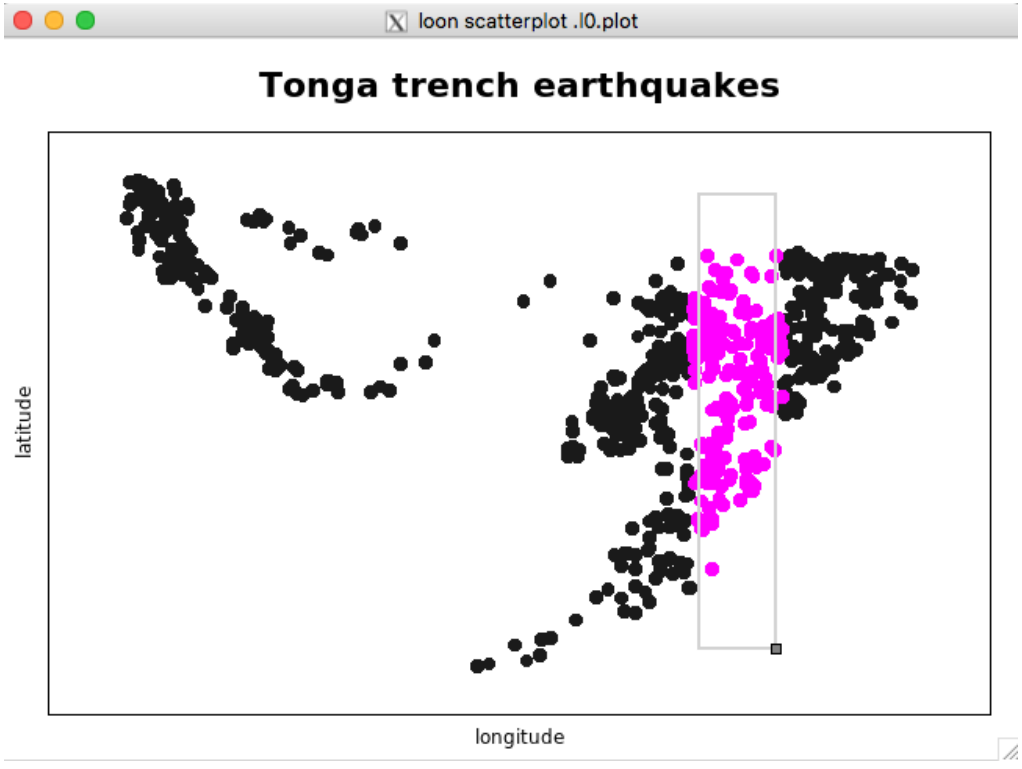




Mouse gestures - brushing

Brushing (selection via a fixed size rectangle)



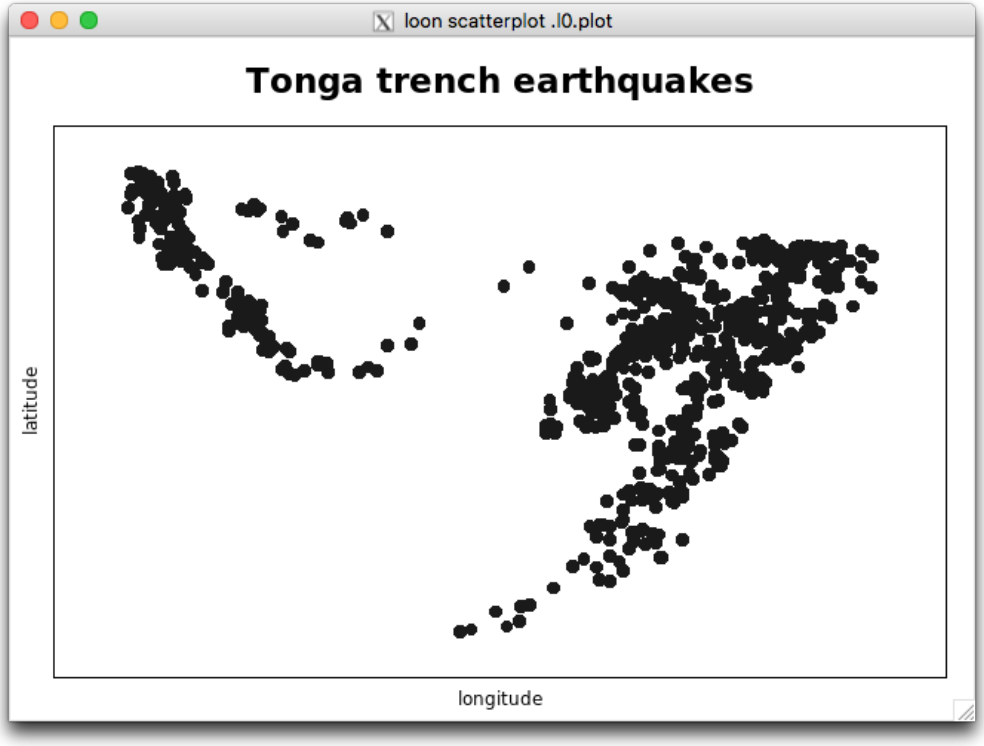


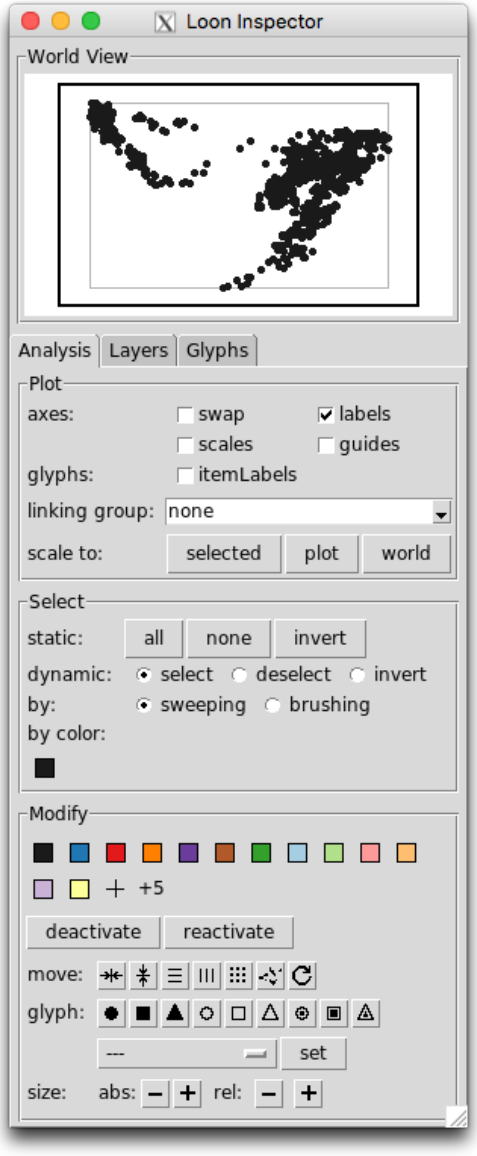
The rectangular "brush"

- maintains its shape as it is moved
- can be reshaped by selecting the small grey square
- tall thin is equivalent to selecting x values
- wide flat to selecting ys

The loon inspector

Whenever a loon plot is active, the **loon inspector** is focussed on that plot:

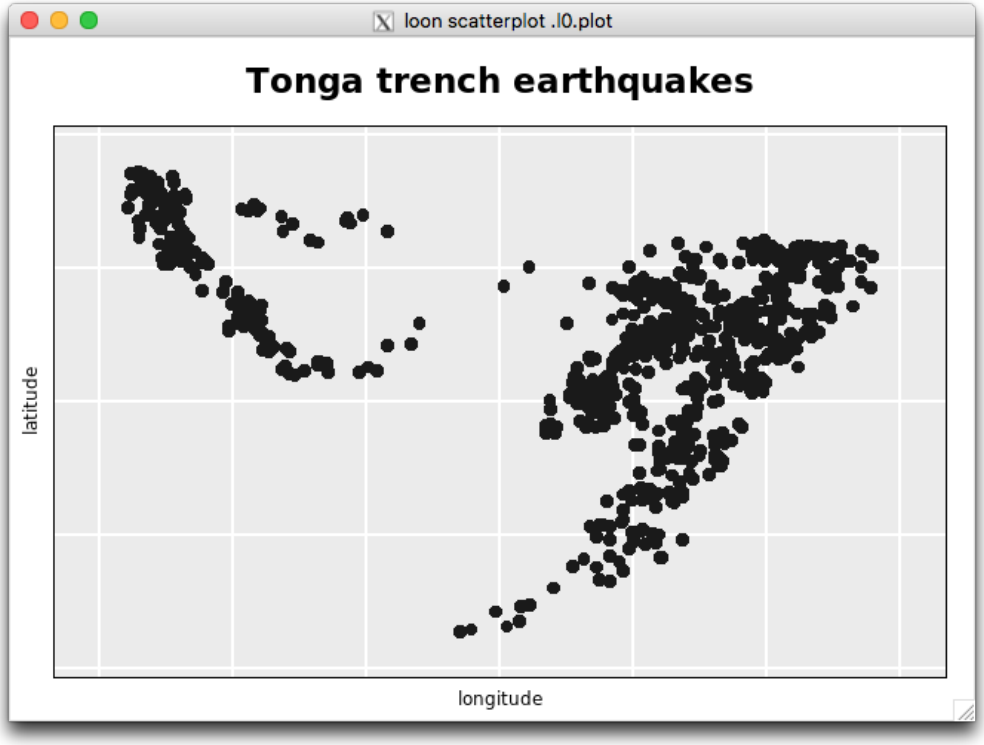


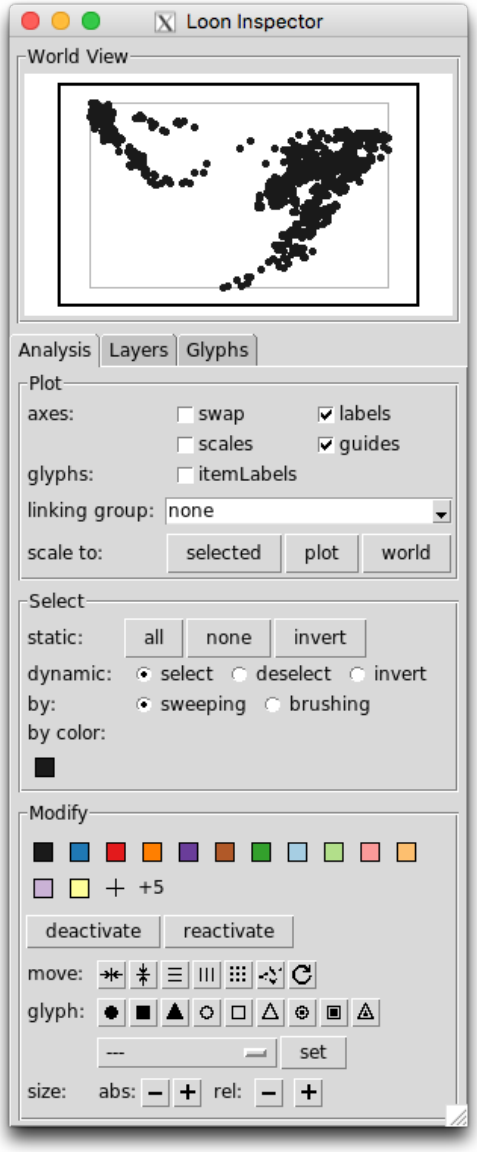


The loon inspector is created when the first loon plot is and it can never be closed while there are any loon plots.

The loon inspector - interacting with the plot

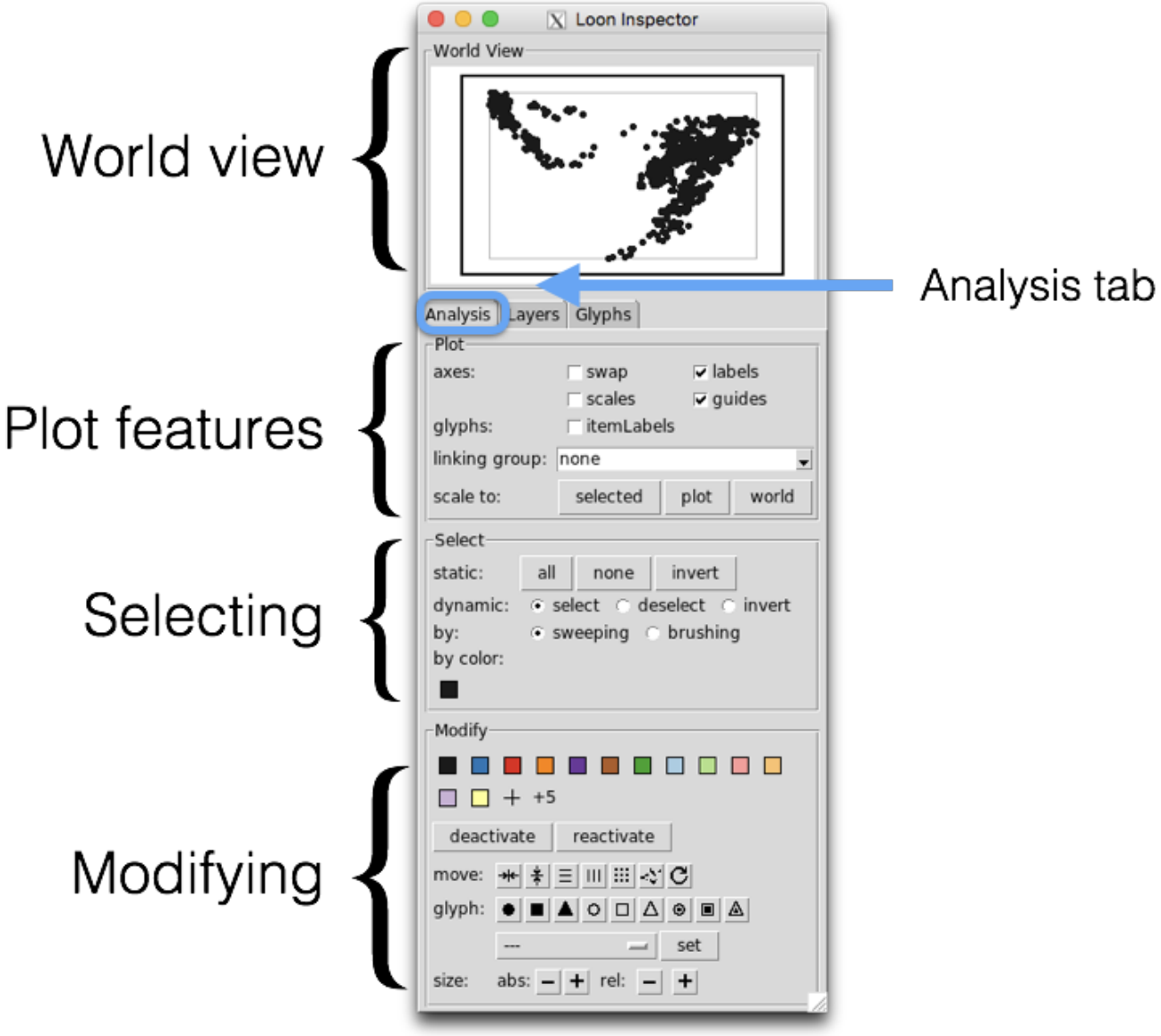
The inspector is used to change the loon plot. For example, checking the "guides" box places a background grid of guide lines on the loon plot.



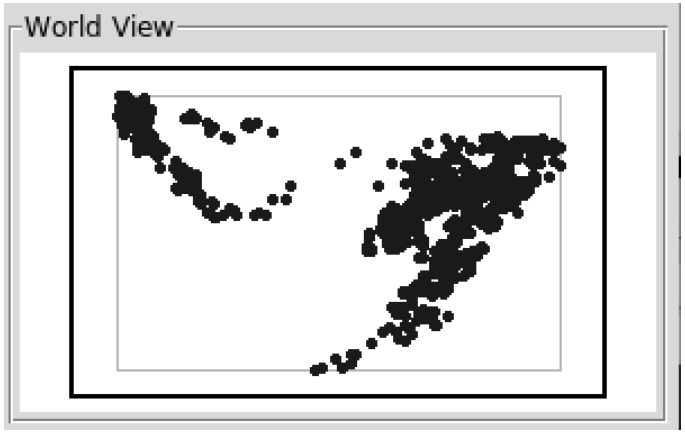


The loon inspector - components

The loon inspector separates into different panels on the analysis tab:



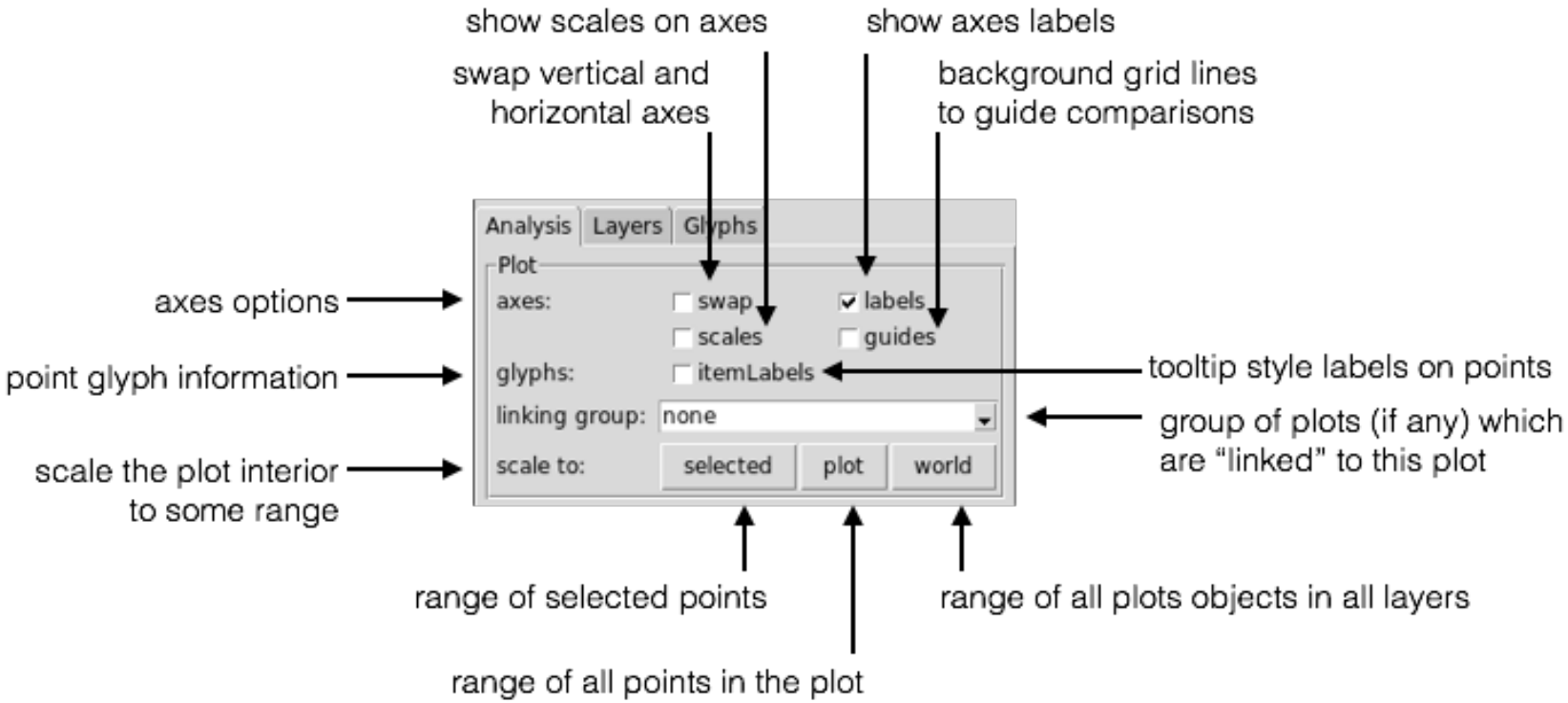
The loon inspector - the World View



The world view always shows the **whole** of the current plot, what is displayed and what is active.

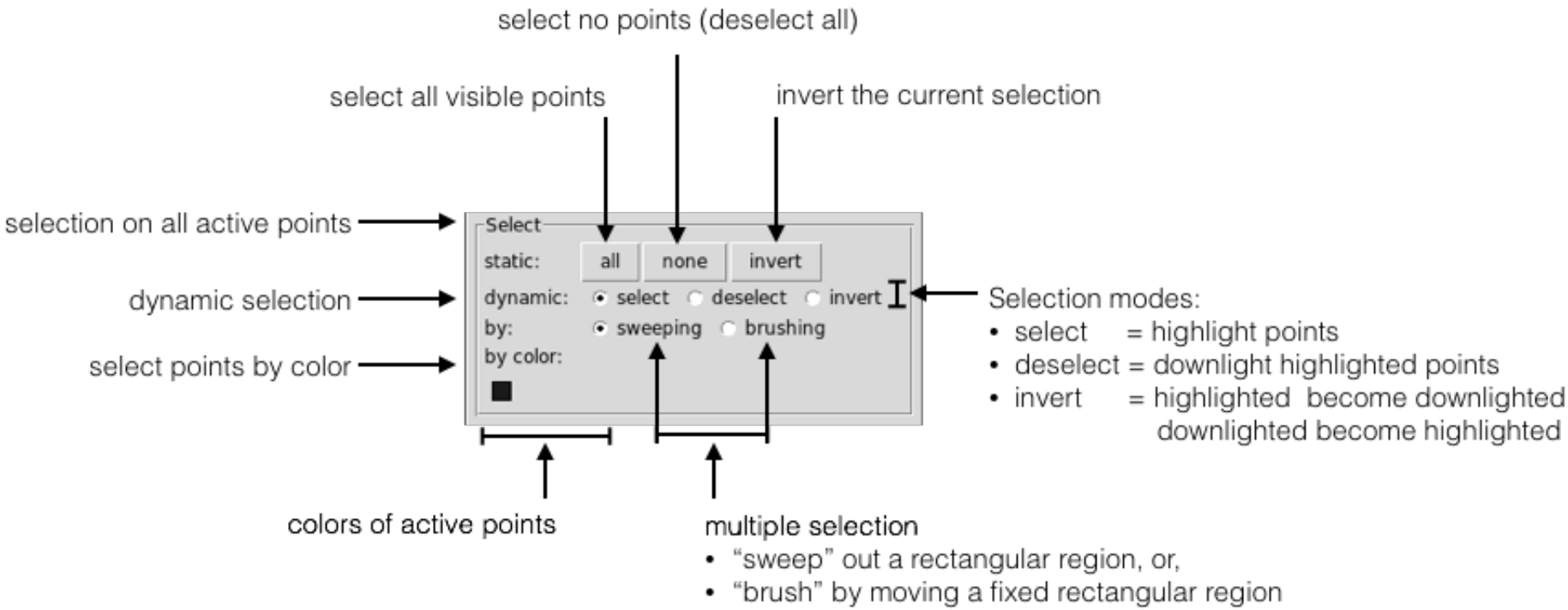
The loon inspector - the Analysis tab

Plot features:



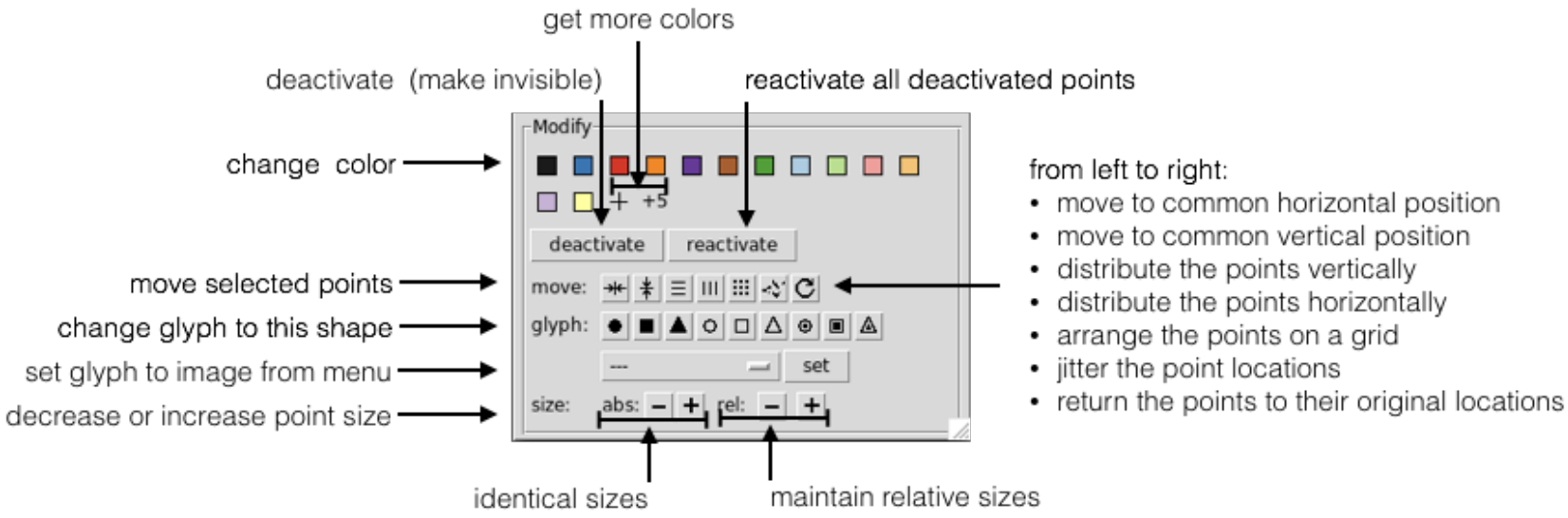
The loon inspector - the Analysis tab

Point selection:



The loon inspector - the Analysis tab

Modifying selected points:

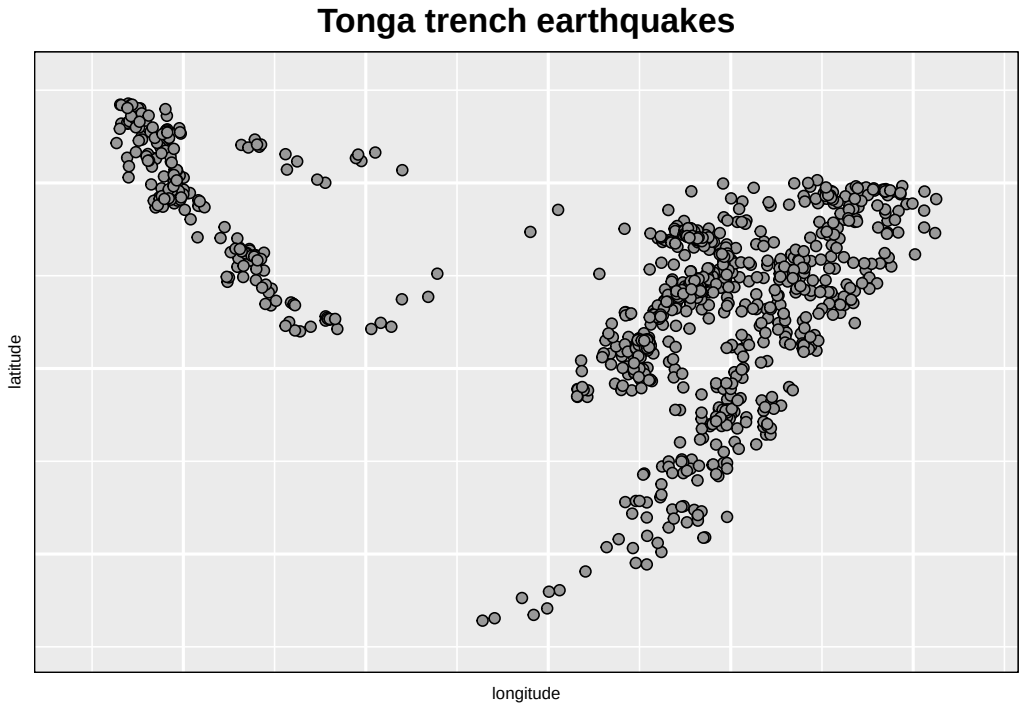


Printing the plot

At any time the **current view** of the loon plot `p` can be transferred to a static `grid` graphics plot simply as `plot(p)` and from there saved as usual. (More on this later.)

For example,

```
plot(p)
```



Adding layers

The plot is a data structure (in `tcltk`) and we can add other plot objects, such as polygons (and other geometric structures), to it.

For example, we can add a map to the current loon plot `p`.

First get the relevant map:

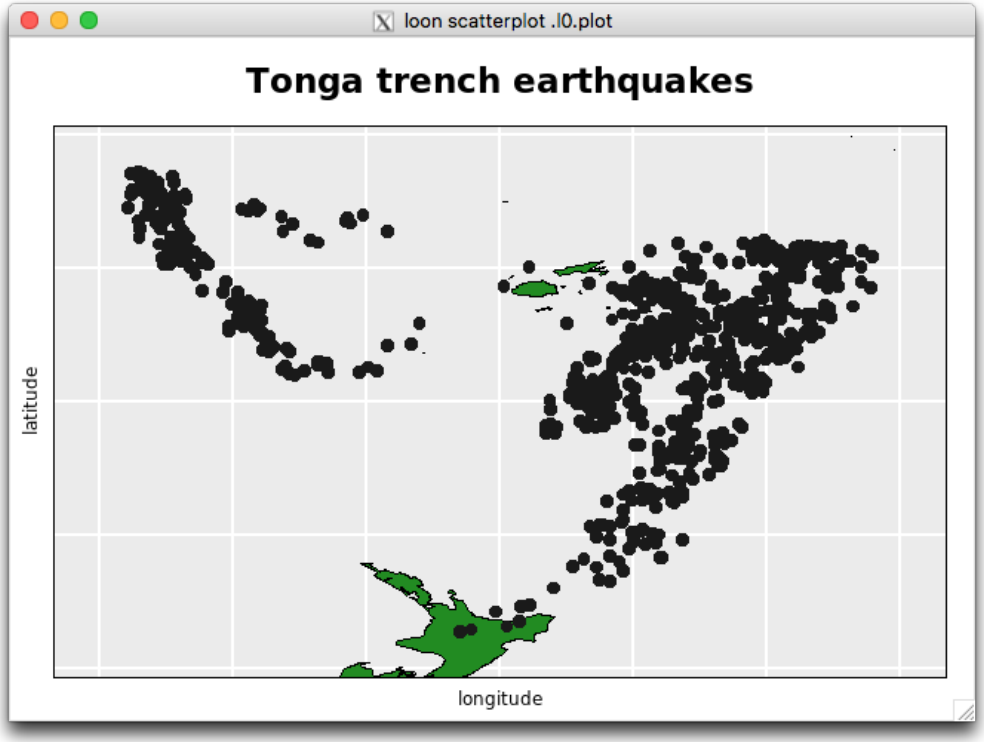
```
library(maps)
NZFijiMap <- map("world2", regions=c("New Zealand", "Fiji"), plot=FALSE)
```

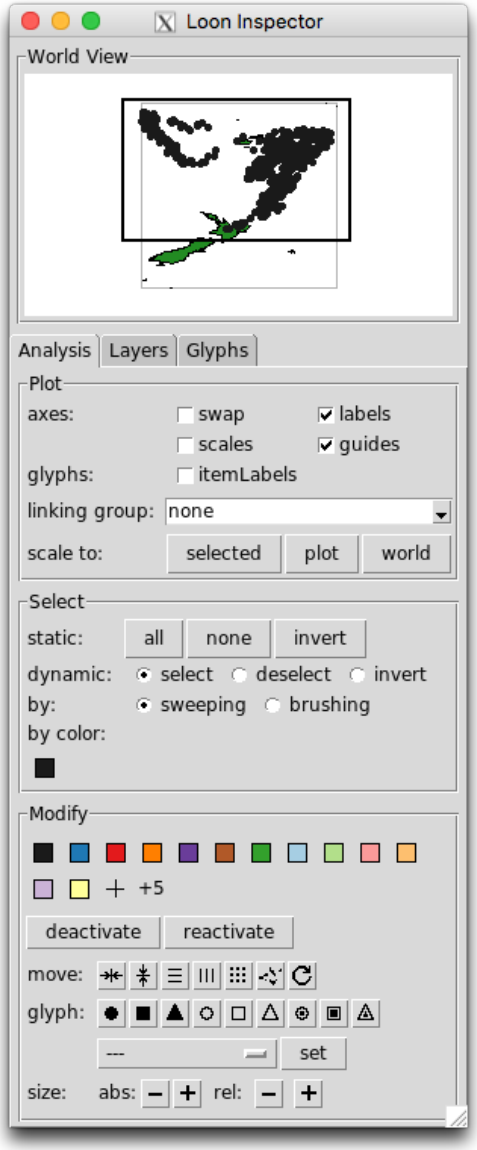
It is added as a "layer" to the loon plot

```
l_layer(p, NZFijiMap,
        label = "New Zealand and Fiji",
        color = "forestgreen",
        index="end")
```

```
## loon layer "New Zealand and Fiji" of type polygons of plot .l0.plot
## [1] "layer0"
```

Adding layers - maps



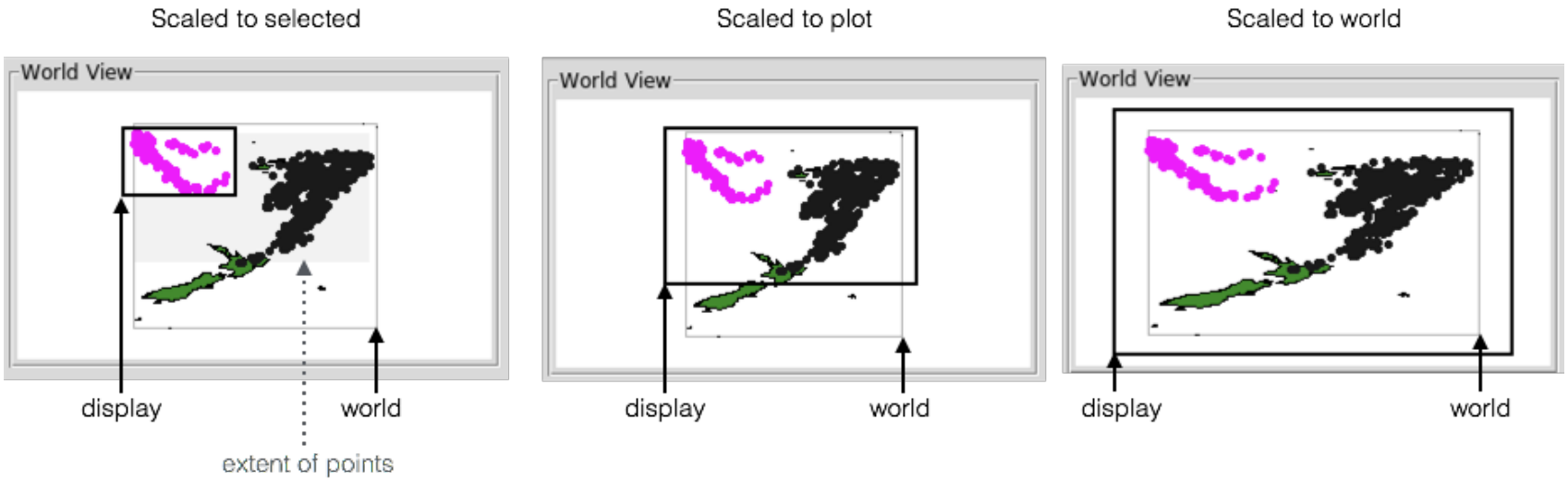


As can be seen in the world view, much of the map is outside of the plot display (shown as the black-bordered rectangle in the world view).

Analysis tab - scaling choices

Adding the map allows us to see the effect of the three plot scaling choices which are available from the inspector.

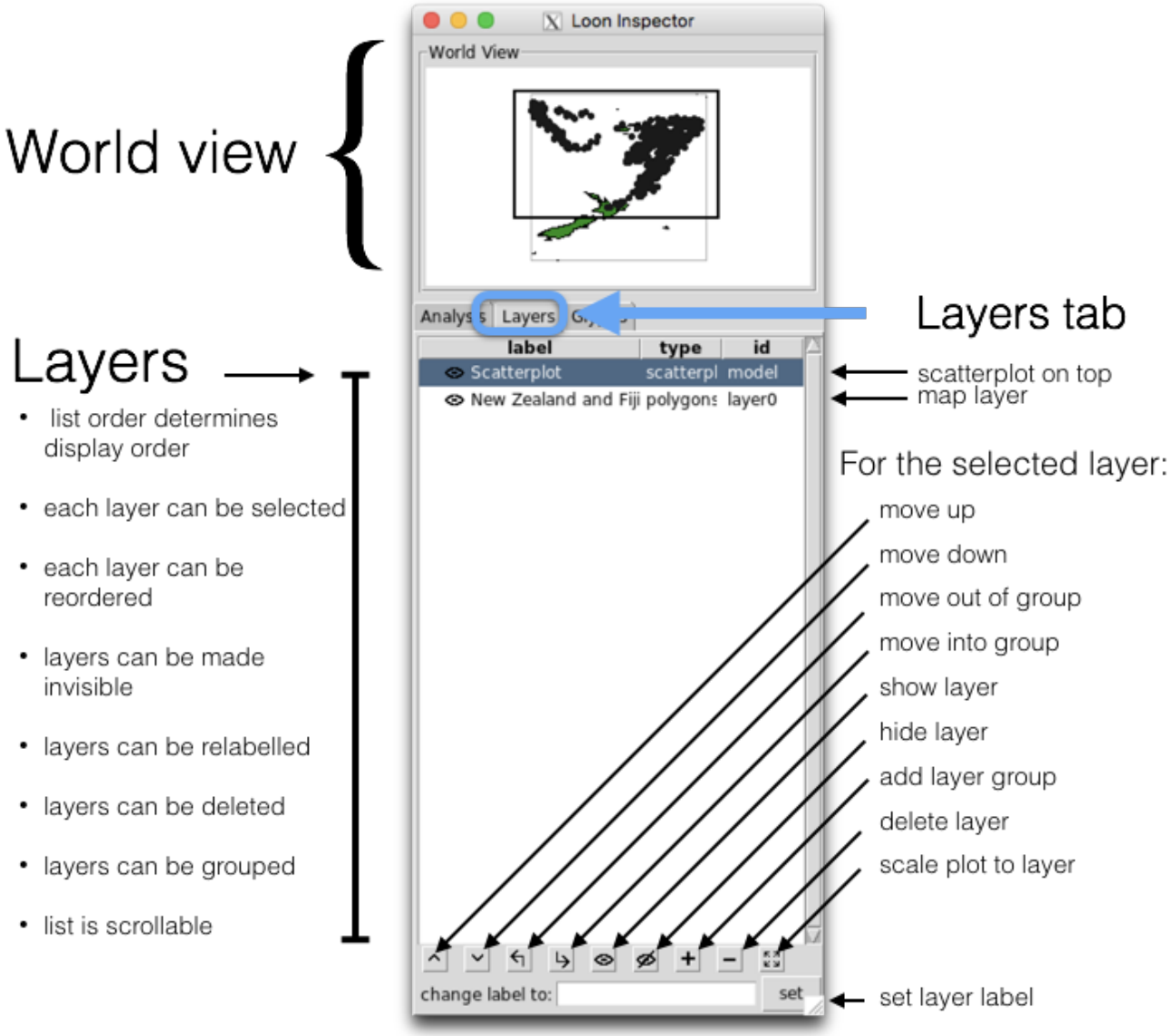
The effect is best seen on the world view:



Note that the plot in the world view matches that of the actual plot. Also the aspect ratio changes with the scaling.

Loon inspector - Layers tab

The map is added as a layer, which can be seen by selecting the "Layers" tab in the inspector:



More than one plot - linking

The quakes data actually contains measurements on several variates:

```
str(quakes)
```

```
## 'data.frame':    1000 obs. of  5 variables:
##  $ lat     : num  -20.4 -20.6 -26 -18 -20.4 ...
##  $ long    : num  182 181 184 182 182 ...
##  $ depth   : int  562 650 42 626 649 195 82 194 211 622 ...
##  $ mag     : num  4.8 4.2 5.4 4.1 4 4 4.8 4.4 4.7 4.3 ...
##  $ stations: int  41 15 43 19 11 12 43 15 35 19 ...
```

We might construct a second plot of the quake magnitude versus its depth:

```
p2 <- l_plot(quakes[,c("depth", "mag")],  ylabel = "magnitude",
             showScales = TRUE, showGuides = TRUE,
             linkingGroup = "quakes")
```

Notes:

- the data are given here as a data frame of two variates
- we specified that both ‘guides‘ and ‘scales‘ be on
- ‘p2‘ is assigned the string "quakes" as its "linkingGroup"

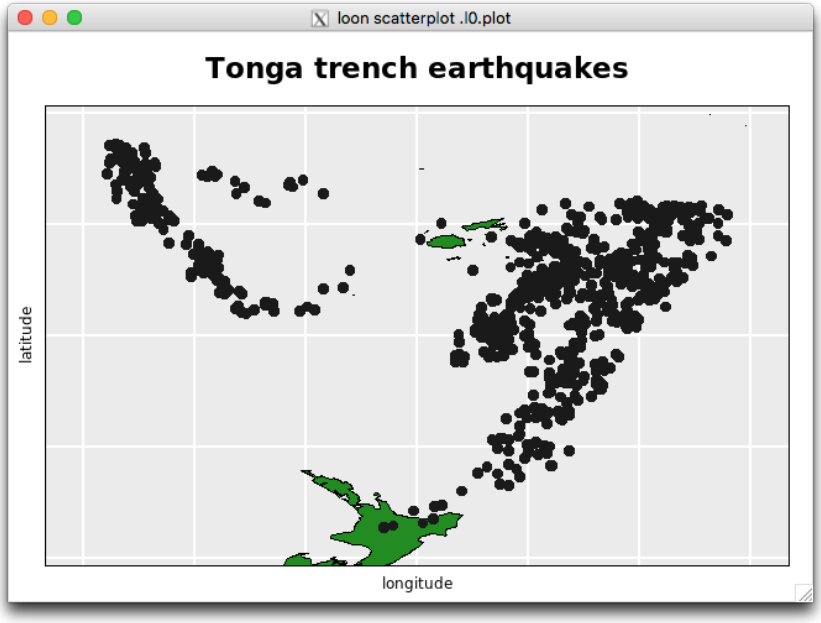
We can ensure that the original p participates in the same linking group by either selecting “quakes” from the inspector for p or by setting p’s linking group directly:

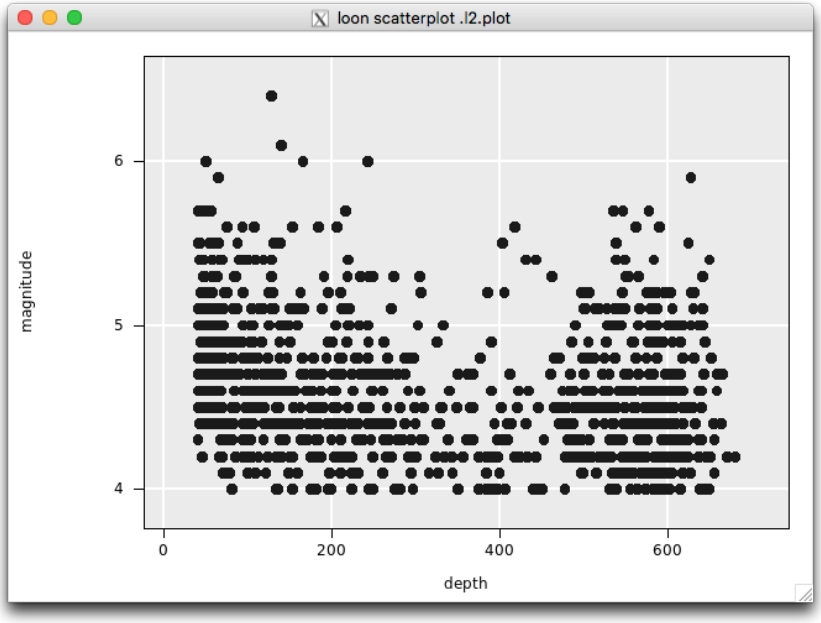
```
l_configure(p, linkingGroup = "quakes", sync="pull")
```

This required a value for sync (i.e. synchronize) which here tells p to “pull” its values from the linking group.

More than one plot - linking

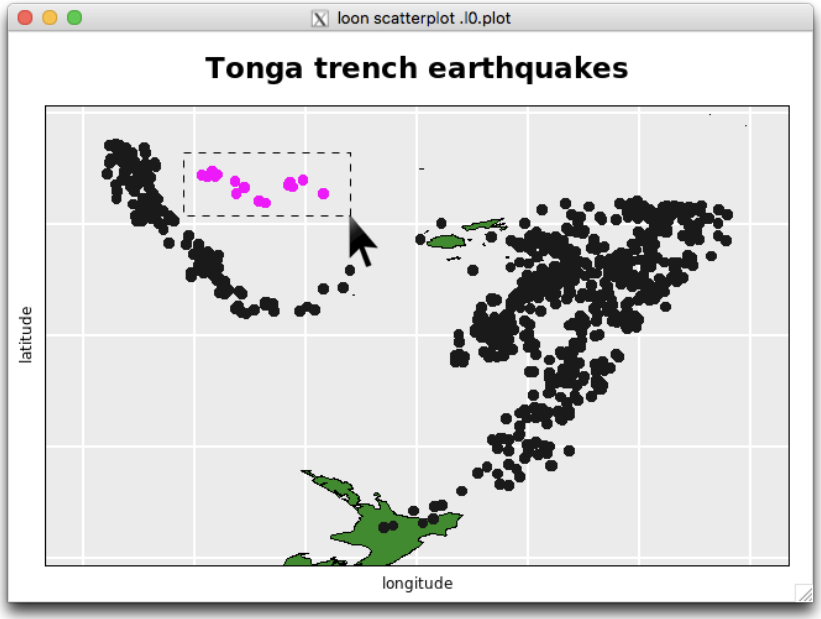
The second plot (p2) shows the quake magnitude versus its depth.

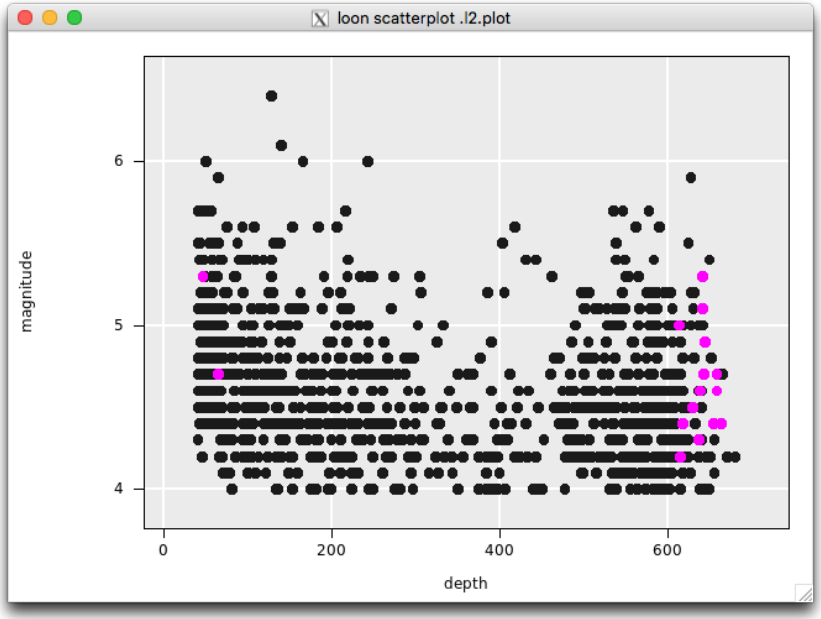




Linking - selection queries

Because the two plots are linked (in the same `linkingGroup`) we can form queries on one plot by using selection (here `sweep`) and see the results (highlighted points) on the second plot:

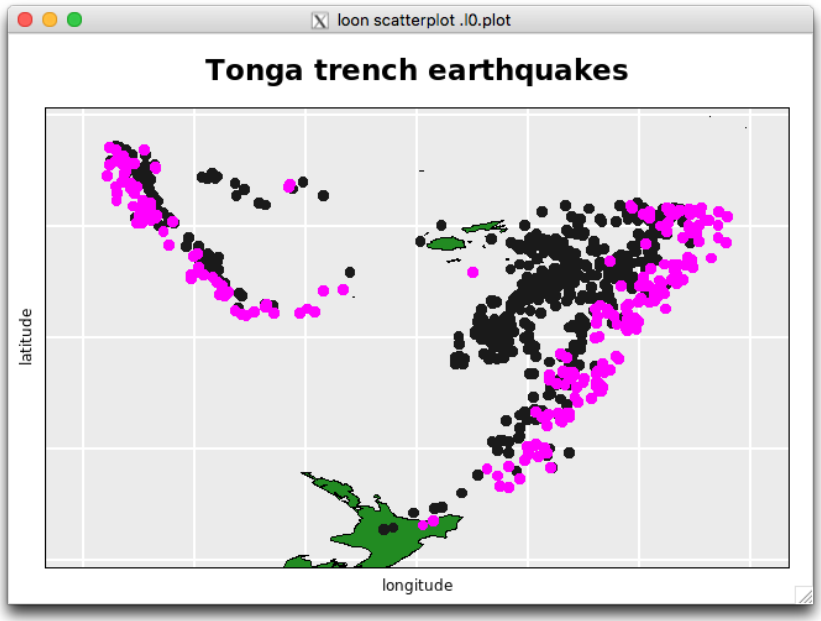


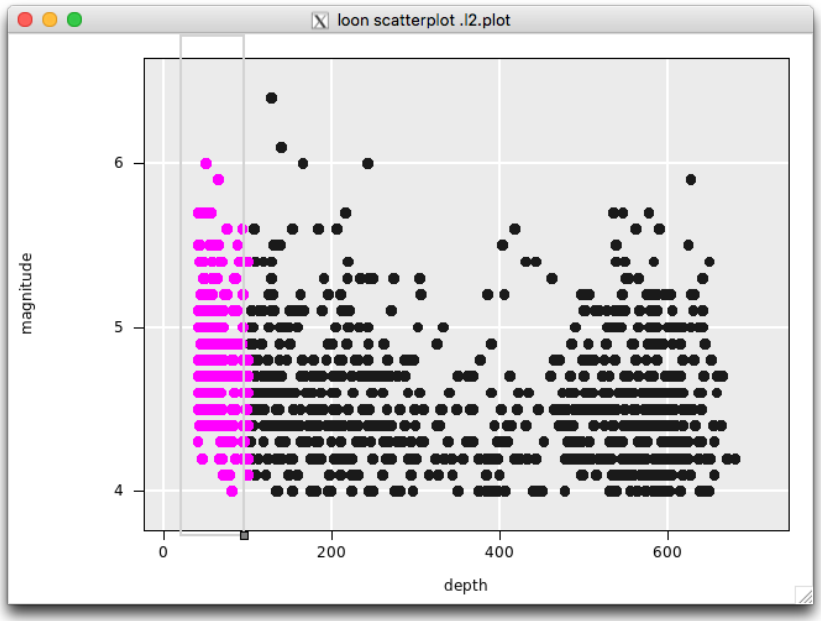


Earthquakes at this location seem to be mostly very deep, with only a couple of relatively shallow quakes. The magnitude of earthquakes in this region seem to relatively spread out (none are amongst the greatest or least magnitude quakes).

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

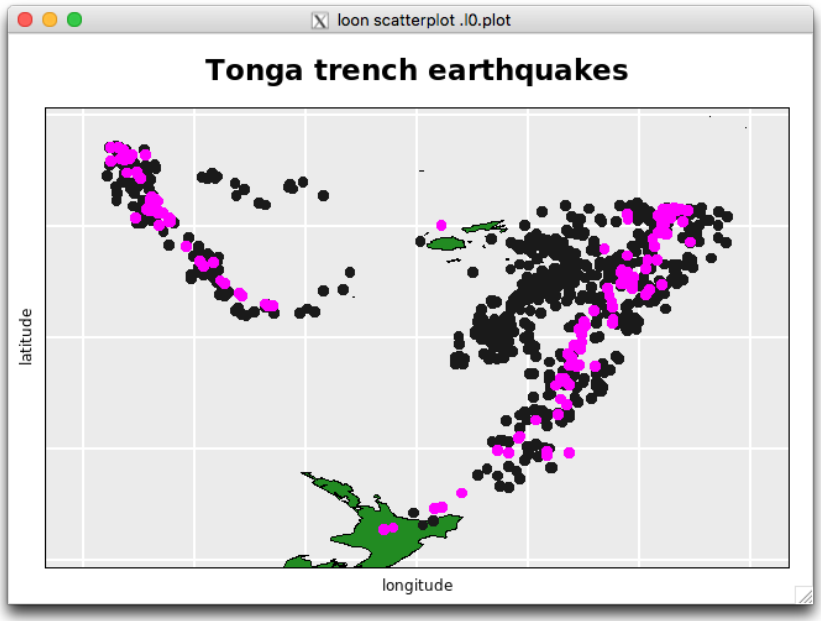


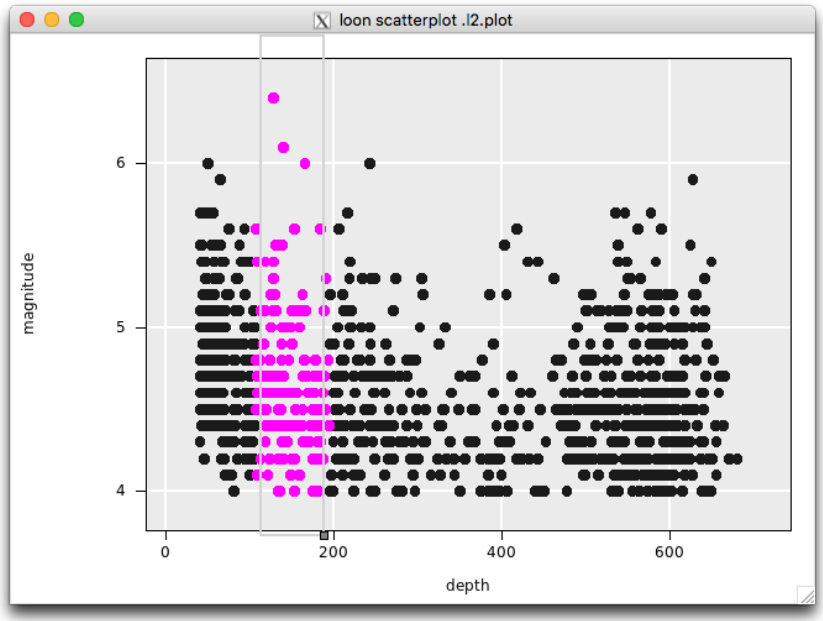


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

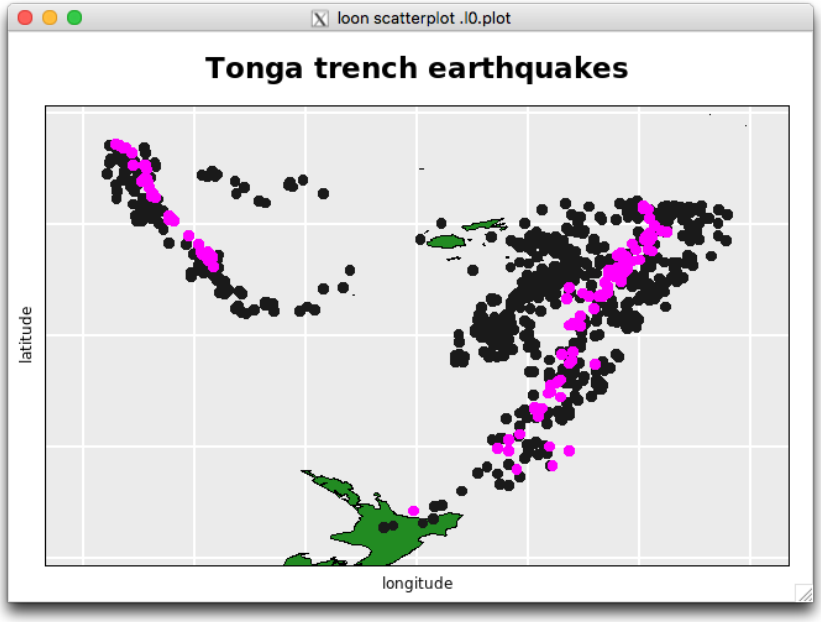


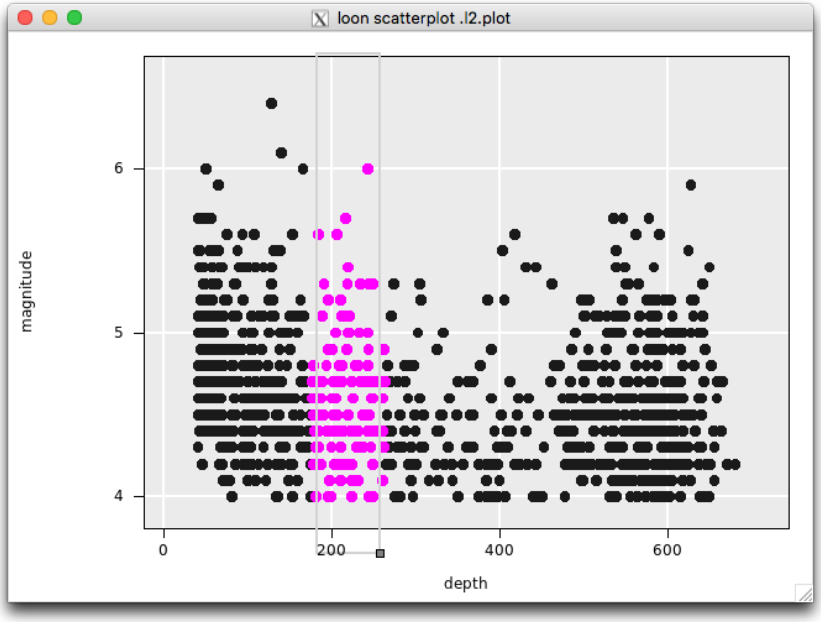


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

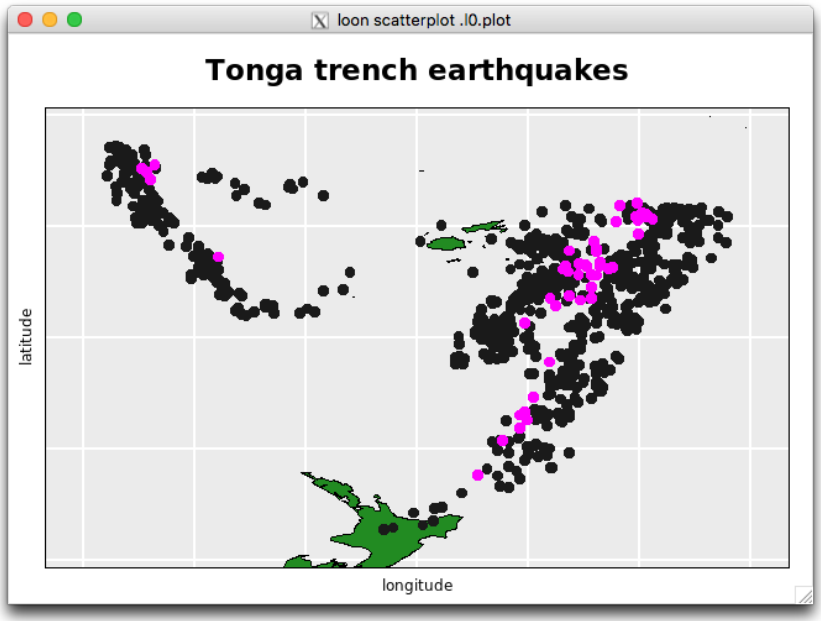


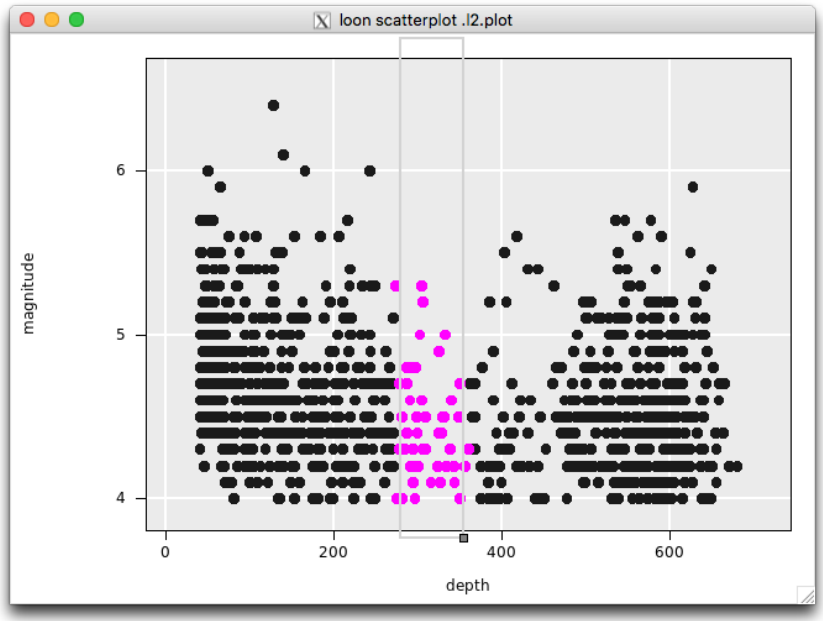


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushin

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

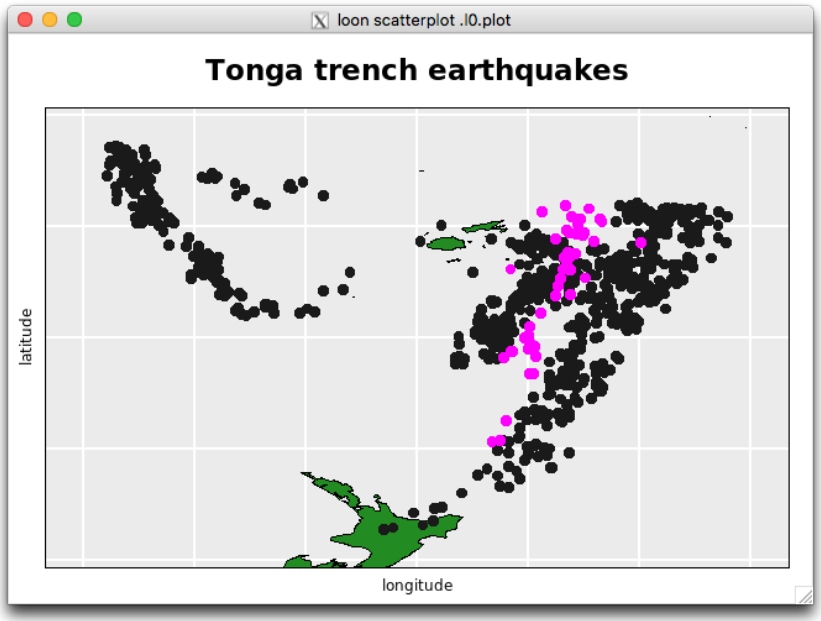


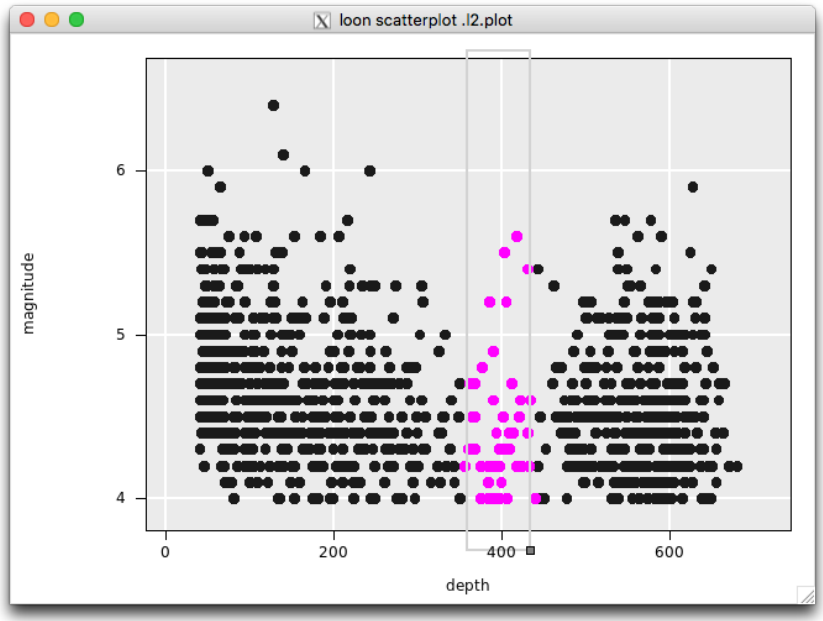


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

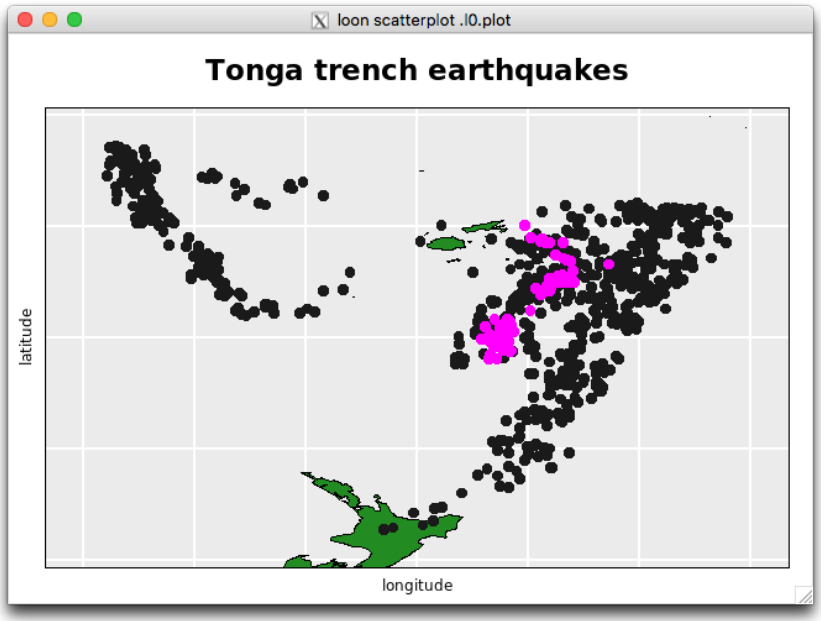


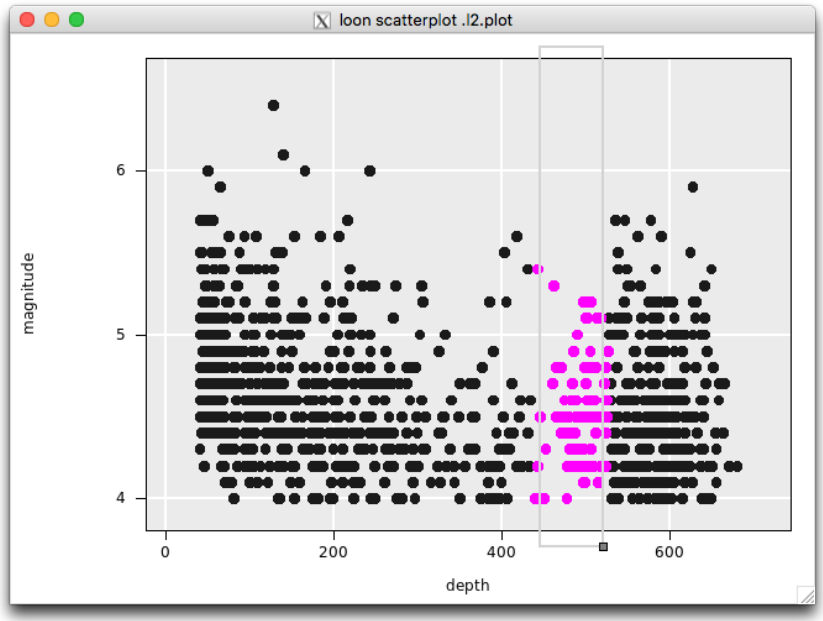


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

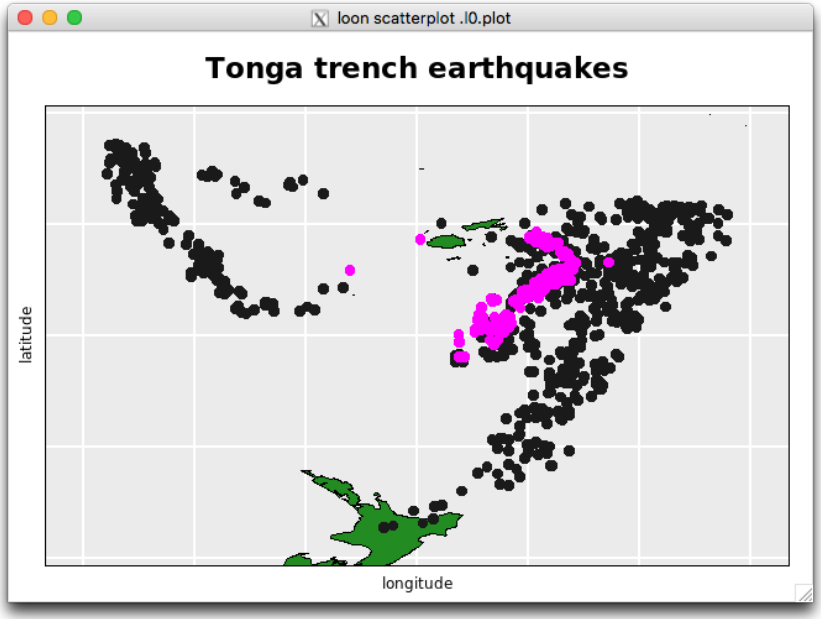


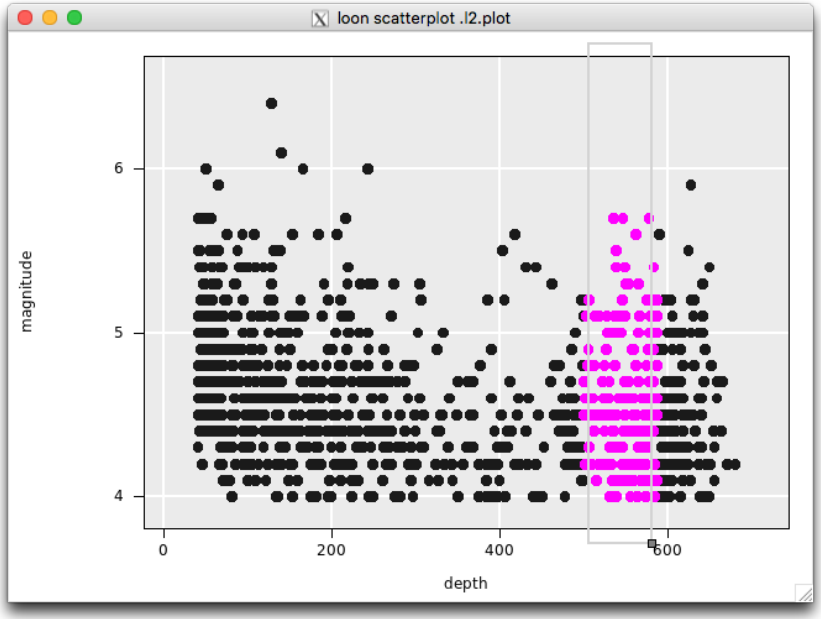


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:

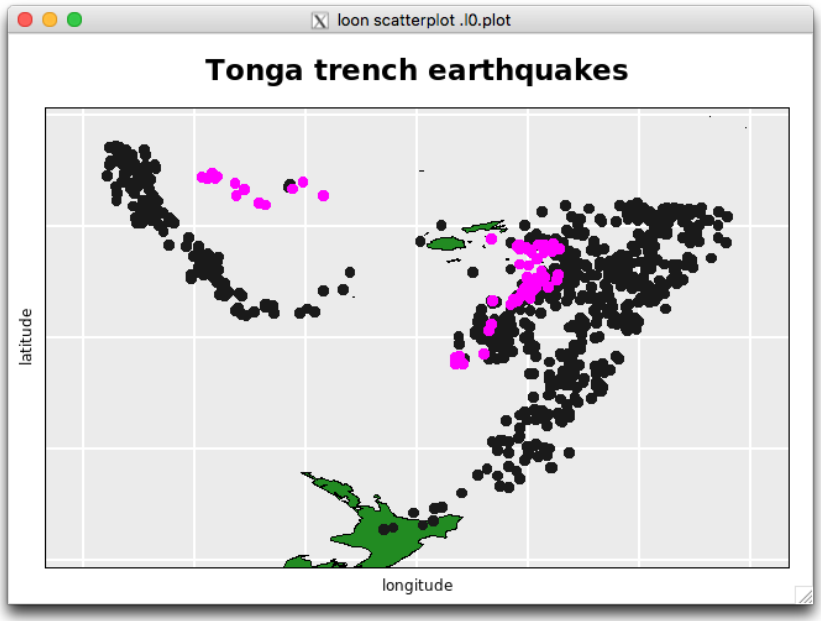


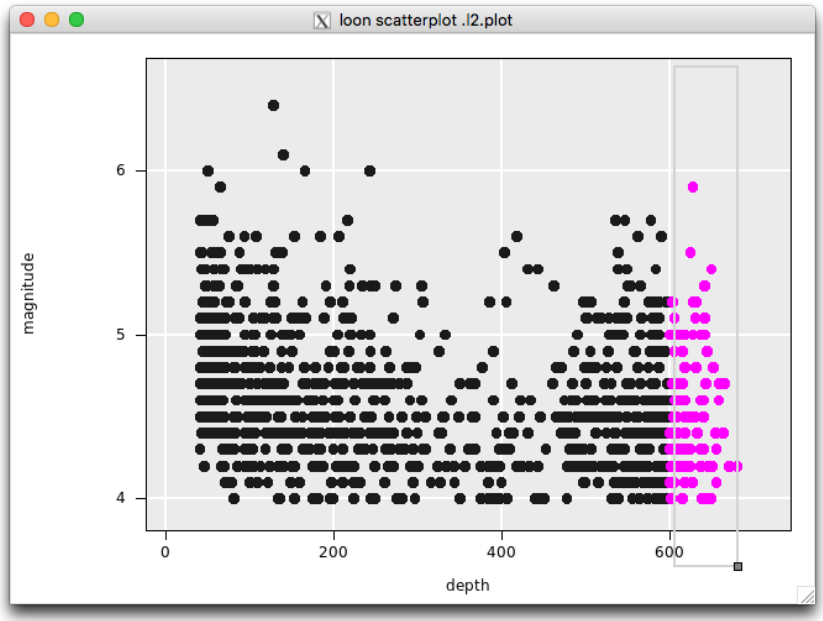


We observe the spatial pattern at left as we brush various depths at right.

Linking - brushing

Alternatively, we could "brush" the magnitude versus depth plot with a tall brush from left to write to see how the locations might change with depth of the quakes:





We observe the spatial pattern at left as we brush various depths at right.

Loon plot states

A `loon` plot has only a single attribute, its class:

```
attributes(p)
```

```
## $class
## [1] "l_plot" "loon"
```

Nevertheless, there are various `tcl` states such as `color` that are associated with a `loon` plot. These may be queried and/or set programmatically.

Because these are `tcltk` data structures, some special functions have been written to access and set these states. These are

```
l_info_states(p)             # gives the names and values of the states on p
names(l_info_states(p))      # returns just the names (not their values)
l_cget(p, "linkingGroup")    # returns the value of the state "linkingGroup"
l_configure(p, color = "steelblue") # sets the color of all points to "steelblue"
```

UNIVERSITY OF WATERLOO

Loon plot states

In R, loon provides some simpler means to achieve the same using more standard R functions. Namely, methods have been written to make the loon states more naturally accessible in R

```
names(p)            # returns just the names (not their values)
```

```
##  [1] "glyph"             "itemLabel"        "showItemLabels"   "linkingGroup"
##  [5] "linkingKey"        "zoomX"            "zoomY"            "panX"
##  [9] "panY"              "deltaX"           "deltaY"           "xlabel"
## [13] "ylabel"            "title"            "showLabels"       "showScales"
## [17] "swapAxes"          "showGuides"       "background"       "foreground"
## [21] "guidesBackground" "guidelines"       "minimumMargins"   "labelMargins"
## [25] "scalesMargins"     "x"                "y"                "xTemp"
## [29] "yTemp"             "color"            "selected"         "active"
## [33] "size"              "tag"              "useLoonInspector" "selectBy"
## [37] "selectionLogic"
```

```
p["linkingGroup"]   # returns the value of the state "linkingGroup"
```

```
## [1] "quakes"
```

```
p["color"] <- "steelblue"  # sets the value of the state linkingGroup to "quakes"
```

Note that for some states like "linkingGroup" more than one value needs to be set simultaneously and this is only achievable using `l_configure()`. For example,

```
# The following may fail when there are other plots in the new group
# because syncing info is also needed
p["linkingGroup"] <- "some other group"
# and this must be passed at the same time as the new group name; more on this shortly
l_configure(p, linkingGroup = "some other group", sync = "pull")
```

Linking - via `color`

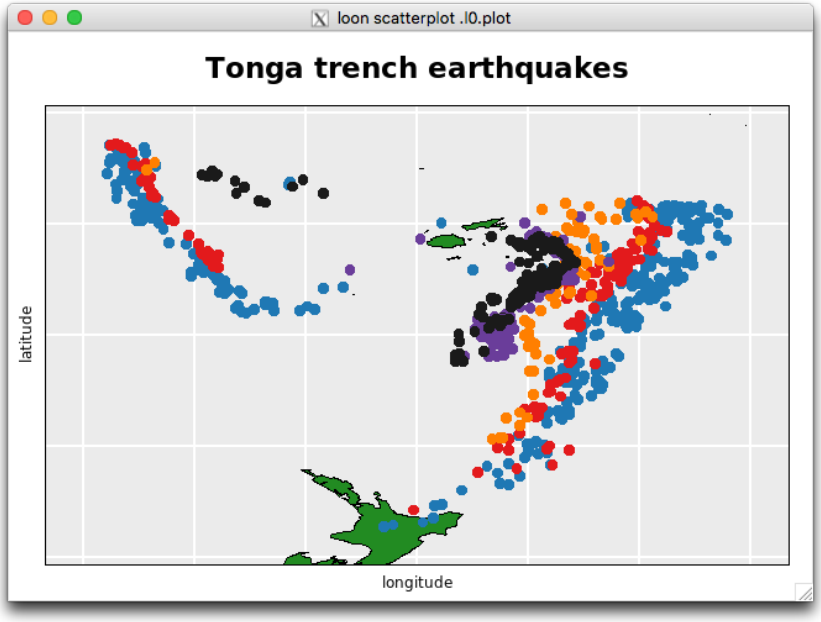
Rather than brush, we might also choose to colour the locations different colours according to the depth. This is easily accomplished either by selecting points and modifying their colour via the inspector, or by directly changing the `color` state of the plot `p`:

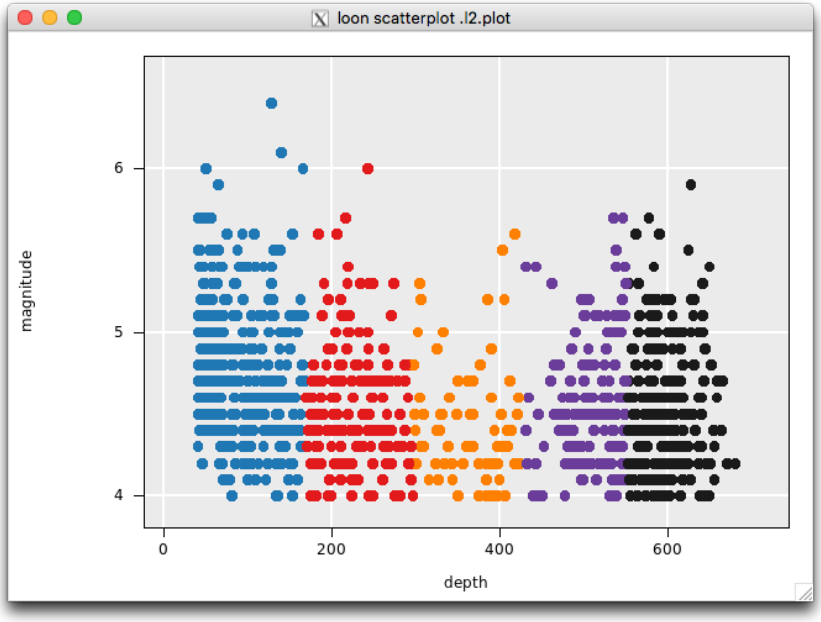
```
# get 5 (equal width) levels by cutting the depths up
depthLevels <- cut(quakes$depth, breaks = 5)
p['color'] <- depthLevels
```

Linking - via `color`

Rather than brush, we might also choose to colour the locations different colours according to the depth. This is easily accomplished either by selecting points and modifying their colour via the inspector, or by directly changing the `color` state of the plot `p`:

```
# get 5 (equal width) levels by cutting the depths up
depthLevels <- cut(quakes$depth, breaks = 5)
p['color'] <- depthLevels
```





Linking - via `color`

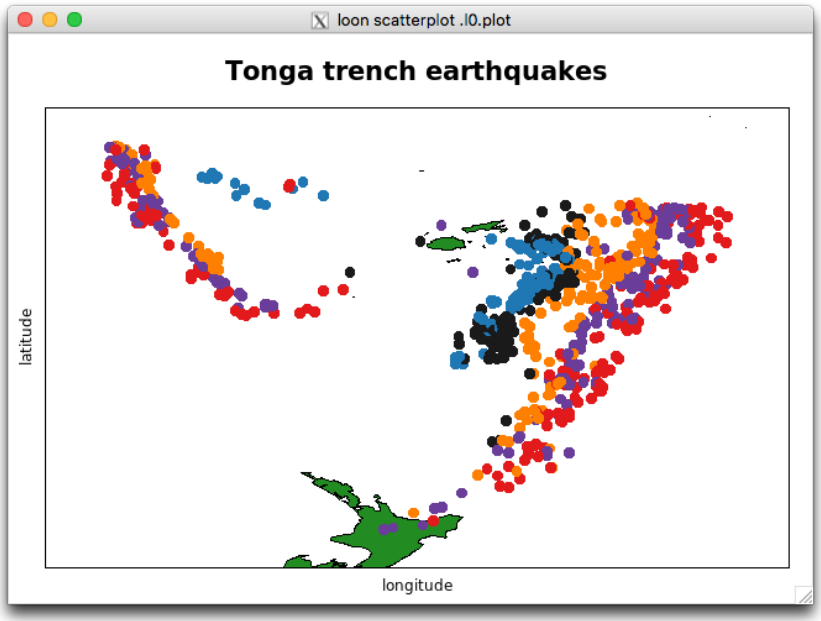
Note that equal count intervals can also be constructed by defining the break points:

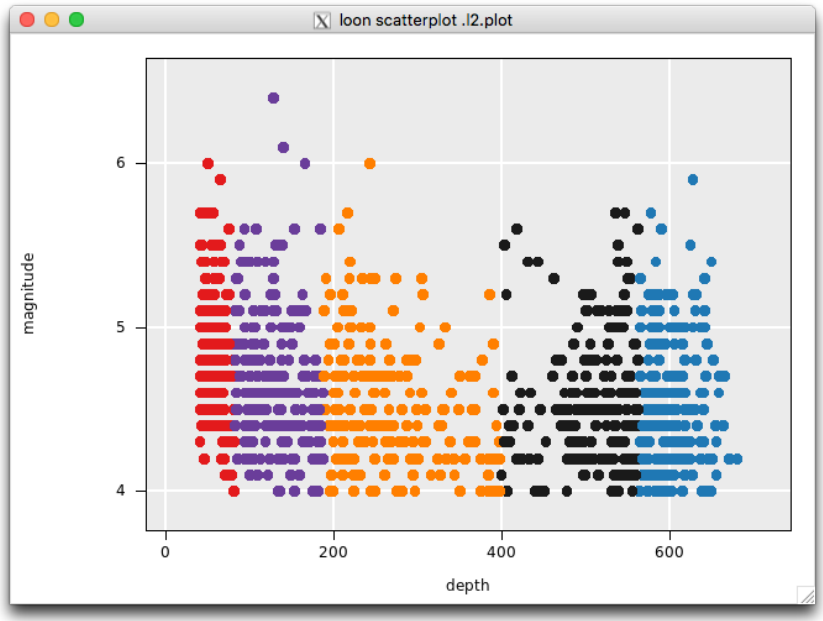
```
quantile_breaks <- quantile(quakes$depth, probs = seq(0, 1, 0.2))
quantile_breaks[1] <- quantile_breaks[1] - 1 # to counter minimum becoming NA
p['color'] <- cut(quakes$depth, breaks = quantile_breaks)
```

Linking - via `color`

Note that equal count intervals can also be constructed by defining the break points:

```
quantile_breaks <- quantile(quakes$depth, probs = seq(0, 1, 0.2))
quantile_breaks[1] <- quantile_breaks[1] - 1 # to counter minimum becoming NA
p['color'] <- cut(quakes$depth, breaks = quantile_breaks)
```





Linking - via `glyph` shape

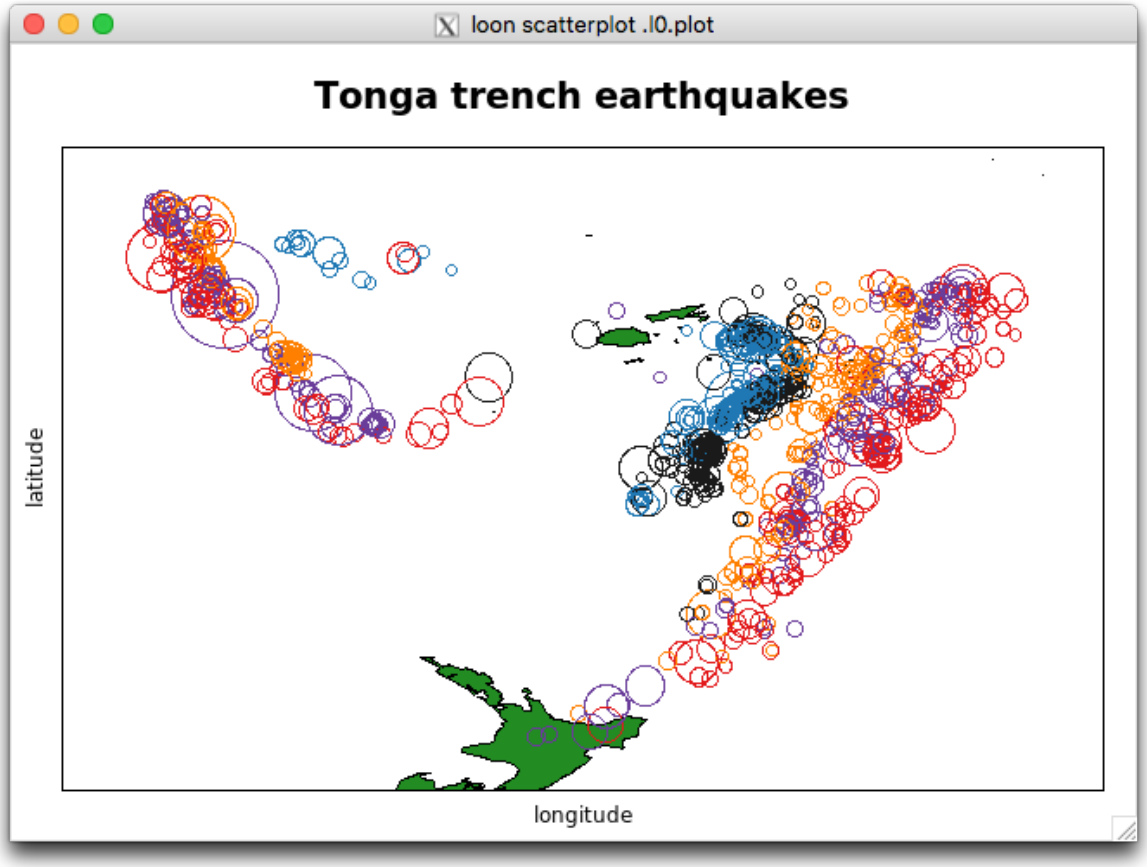
Similarly, we could change the shape of the points, that is its `glyph`, or its size programmatically:

```
p["glyph"] <- "ocircle"    # for "open circle"
sizeByMagnitude <-  (10 ^ quakes$mag) / 10000  # N.B. Richter scale is log scale
sizeByMagnitude <- 2 + sizeByMagnitude - min(sizeByMagnitude)
p["size"] <- sizeByMagnitude
```

Linking - via `glyph` shape

Similarly, we could change the shape of the points, that is its `glyph`, or its size programmatically:

```
p["glyph"] <- "ocircle"    # for "open circle"
sizeByMagnitude <-  (10 ^ quakes$mag) / 10000  # N.B. Richter scale is log scale
sizeByMagnitude <- 2 + sizeByMagnitude - min(sizeByMagnitude)
p["size"] <- sizeByMagnitude
```

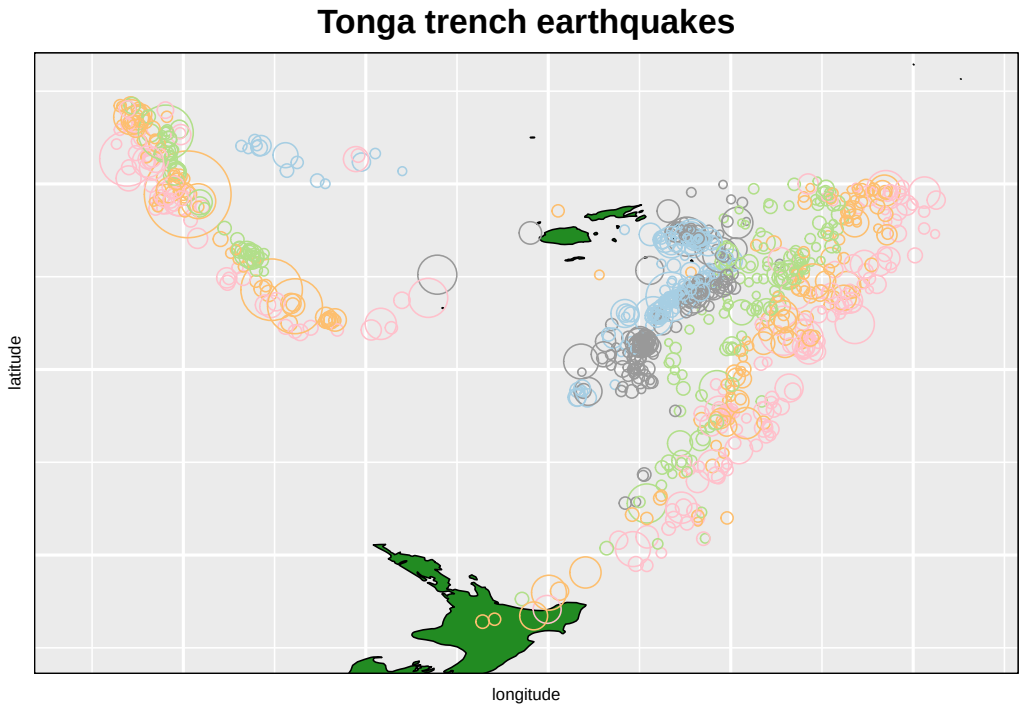


Printing the plot

Note again, that **at any time** the **current view** of the loon plot `p` can be transferred to a static `grid` graphics plot simply as `plot(p)`.

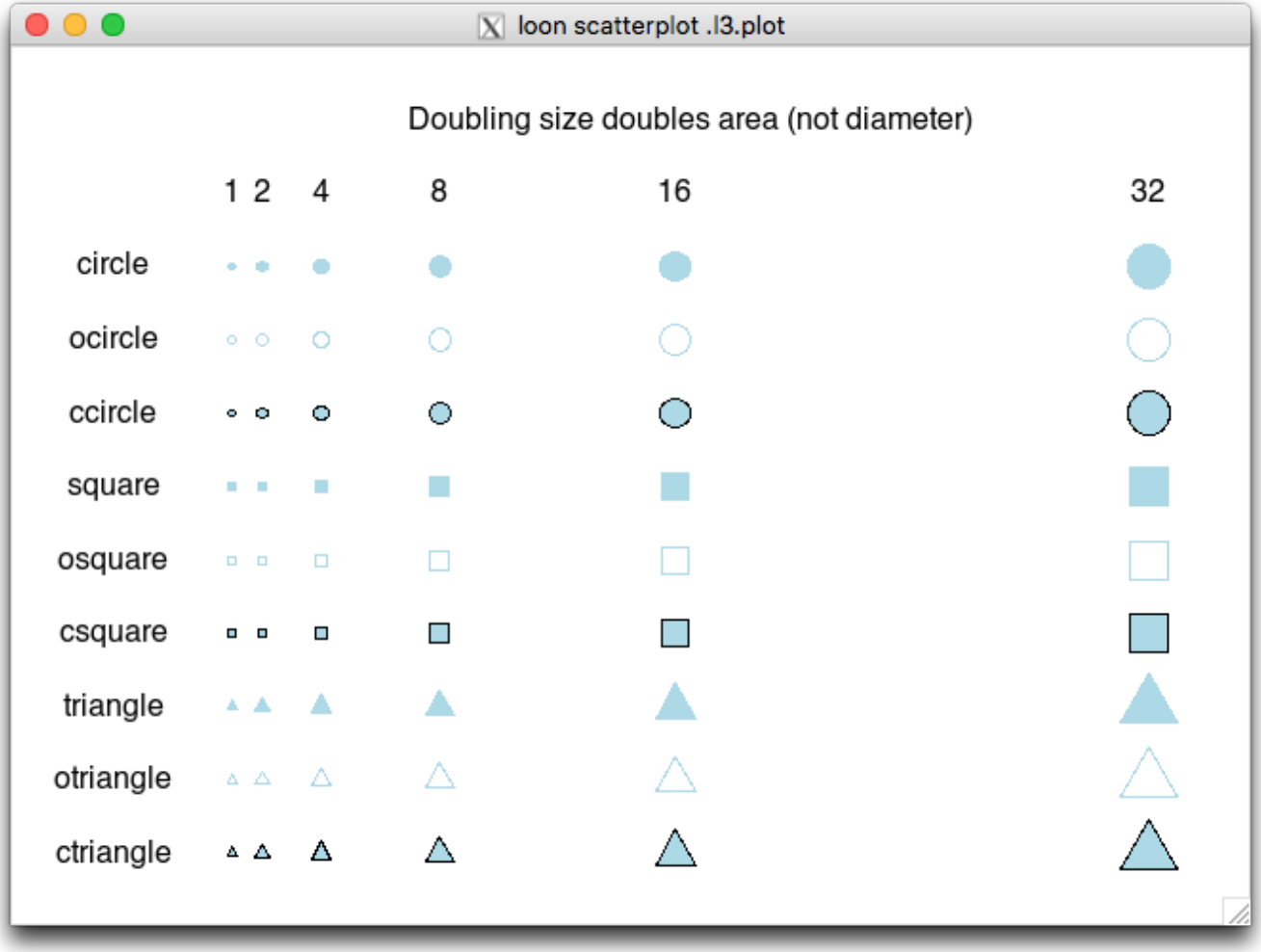
For example, `p` now looks like

```
plot(p)
```



Point glyphs - shapes and sizes

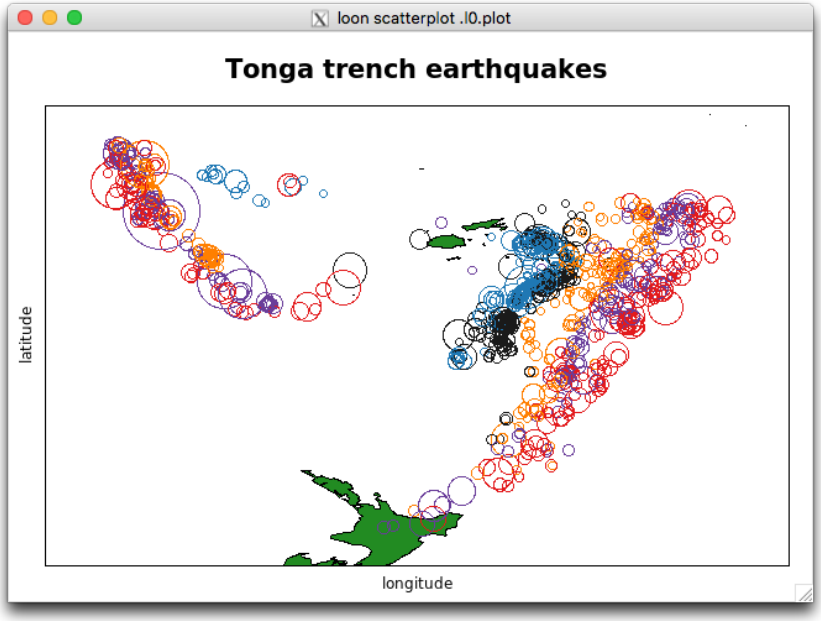
There are numerous different shapes (and sizes) to choose from:

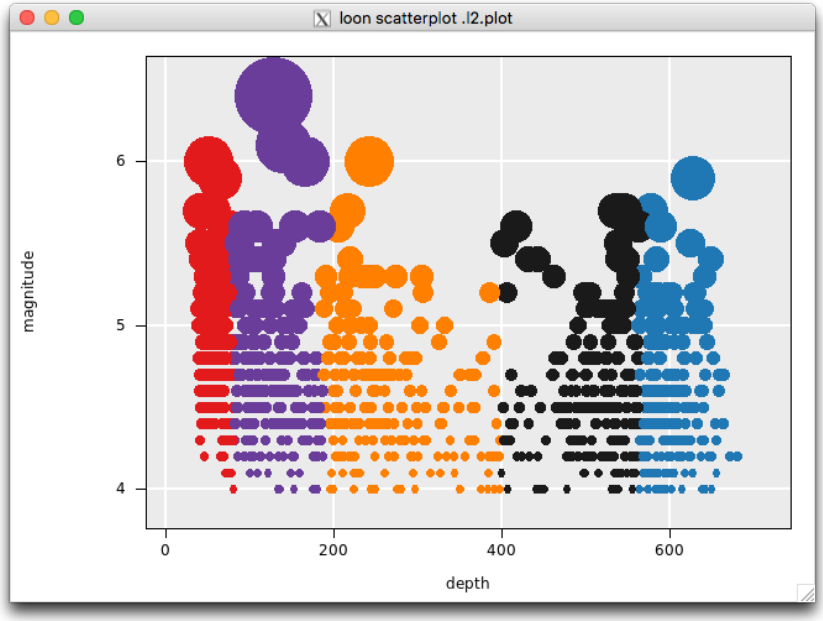


Again, `size` refers to the relative (for each shape) area of the glyph and does not mean its linear extent (or diameter). (To change the size dramatically, as if by linear extent, then the user might change the size by squaring.)

Linking - linked states

A glance at the second scatteplot shows that, like colour, the size of the glyph changed there because it was linked to the first scatterplot. However the shape of the glyph did **not**.





The states which are linked to, and hence eligible to change with, the plots participating in the same `linkingGroup` are unique to each plot:

```
l_getLinkedStates(p)
```

```
## [1] "color"    "selected" "active"   "size"
```

Linking - linked states

Plots in the same `linkingGroup` indicate which of their states, they are willing to have change with the group. These linked states can be queried via `l_getLinkedStates()` as before:

```
l_getLinkedStates(p)
```

```
## [1] "color"    "selected" "active"   "size"
```

Linked states can be updated for any plot using `l_setLinkedStates()` as in:

```
l_setLinkedStates(p, c(l_getLinkedStates(p), "glyph"))
l_setLinkedStates(p2, c(l_getLinkedStates(p2), "glyph"))
```

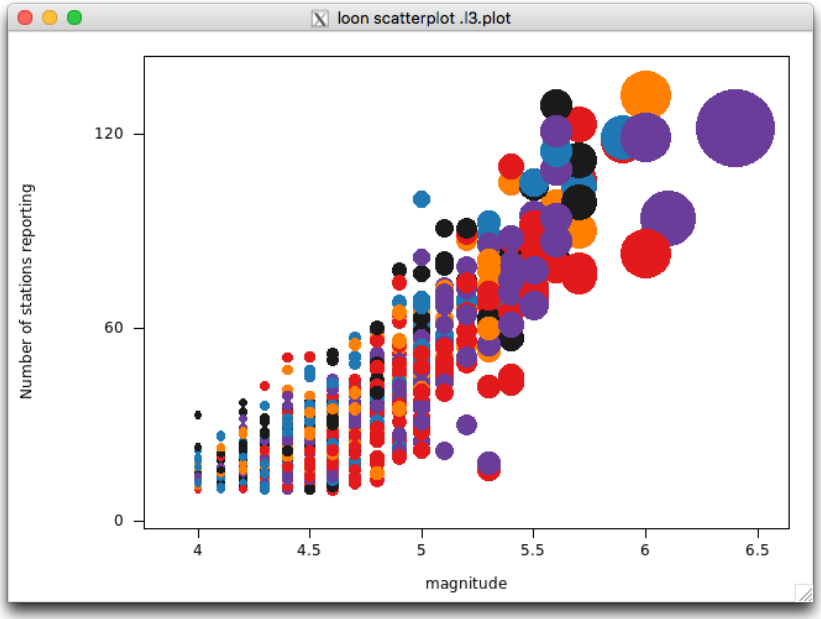
As a consequence each of `p` and `p2` will now change its `glyph` whenever the other does (note that this requires the glyph to actually change before that change is propagated). This is because they are members of a common `linkingGroup` quakes, which can be queried (and set) via `p['linkingGroup']`.

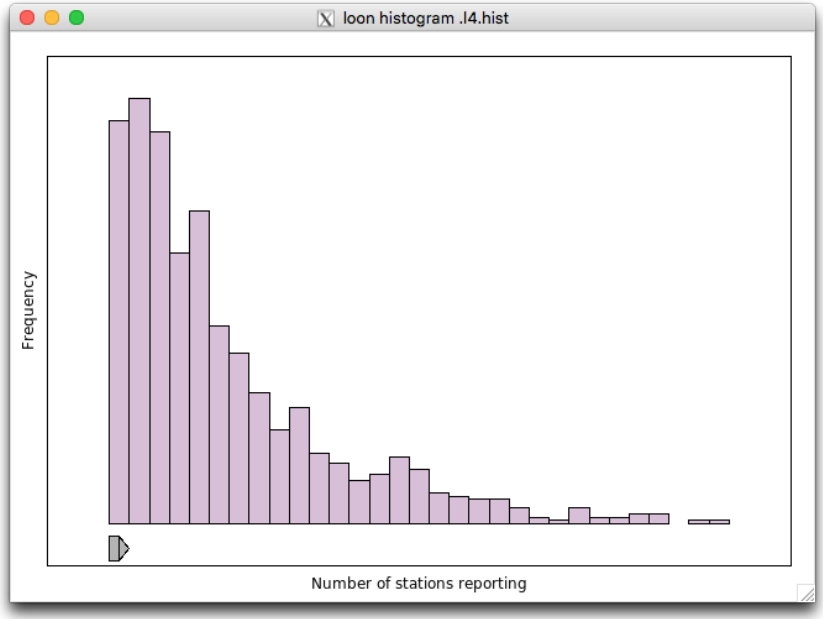
Should another plot, say `p3`, be added to the same `linkingGroup`, then the state `glyph` for `p3` will not change whenever that of either `p` or `p2` does unless `p3` has also registered its `glyph` state as a linked one. Only those linked states that are common between plots in the same `linkingGroup` will be linked.

Plots and histograms via `l_hist()`

We could introduce other plots that are linked to the previous plots

```
p3 <- l_plot(quakes$mag, quakes$stations,
             xlabel = "magnitude", ylabel = "Number of stations reporting",
             showScales = TRUE, linkingGroup = "quakes")
h <- l_hist(quakes$stations, xlabel = "Number of stations reporting",
            linkingGroup = "quakes")
```



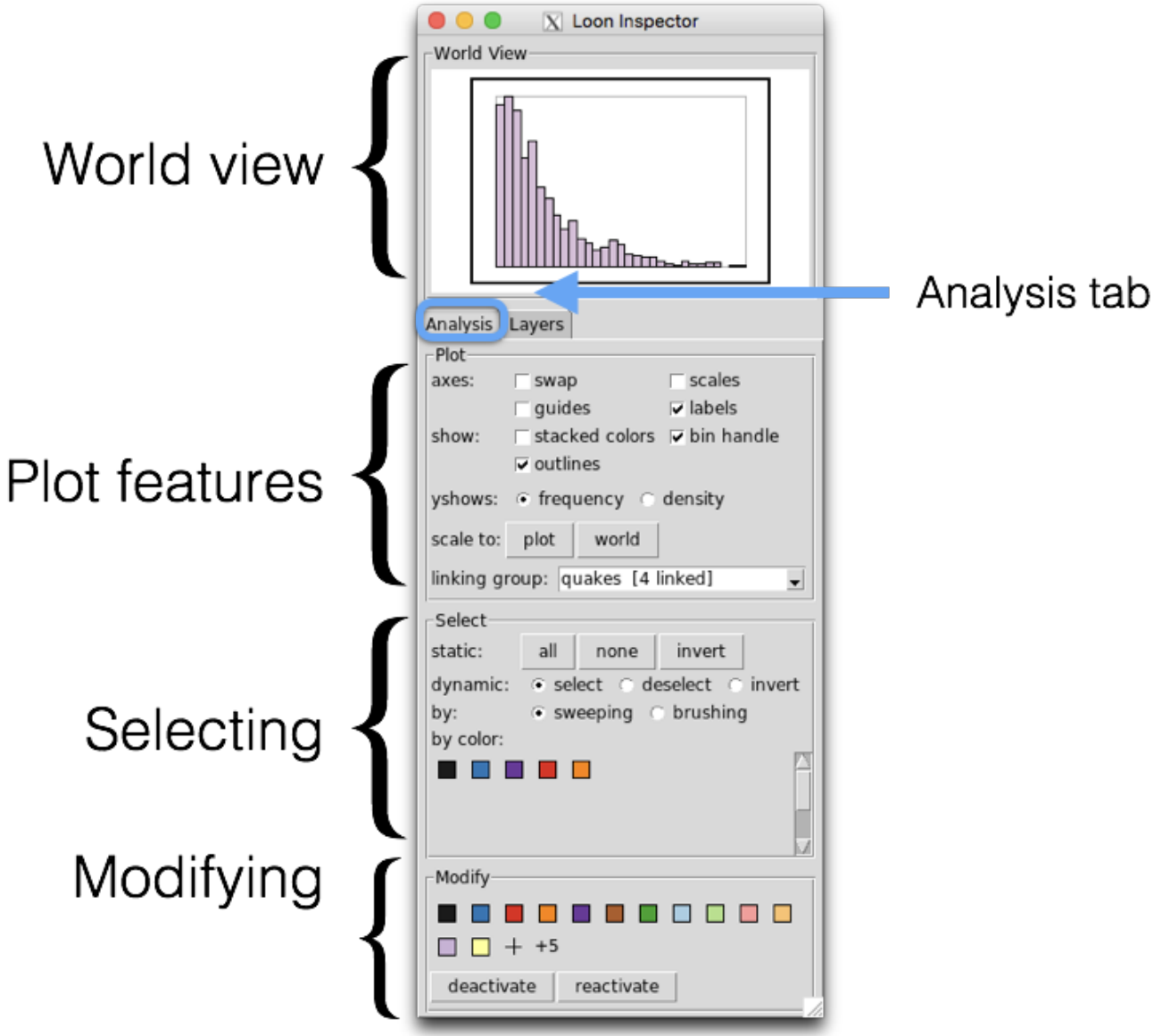


Note:

- the strong positive relationship between the number of stations reporting a quake and its magnitude
- the size of the circle is taken from the common `linkingGroup` but the shape is not
- the colours associated with depth appear in the scatterplot but not in the histogram

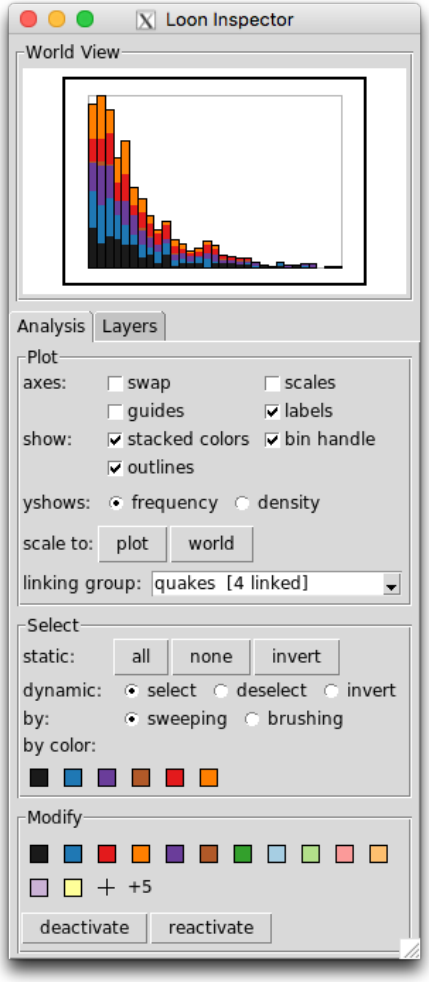
Loon inspector for a histogram - Analysis tab

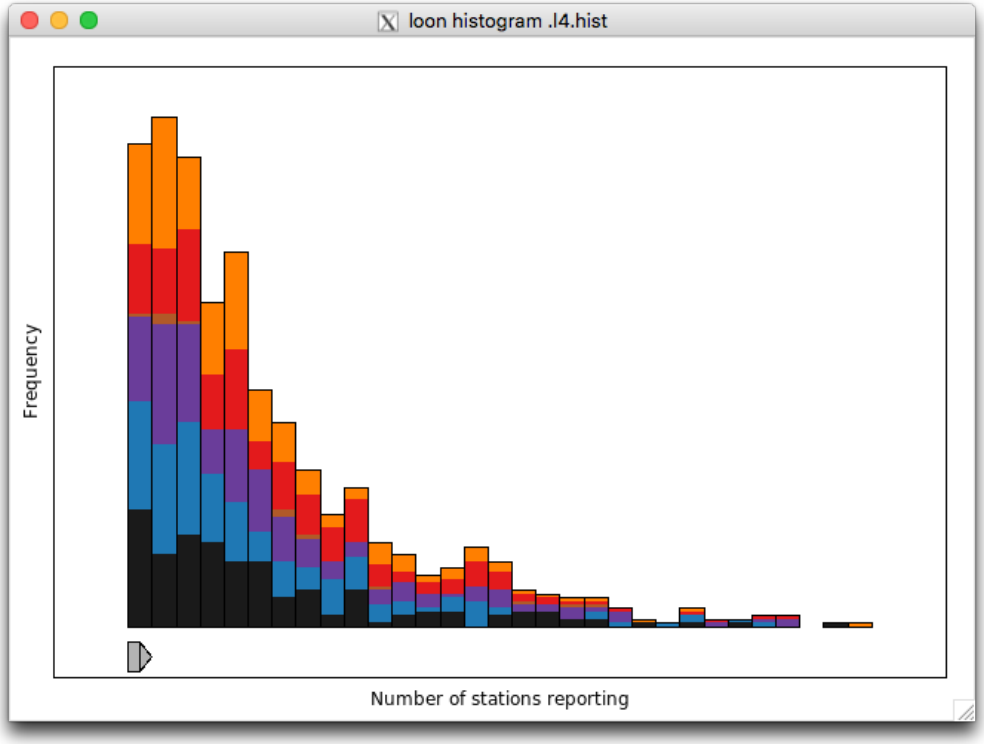
When the histogram is the active window, the inspector changes its focus to the histogram, now with panels specialized for a histogram on the analysis tab:



Analysis tab - `stacked colors` check box

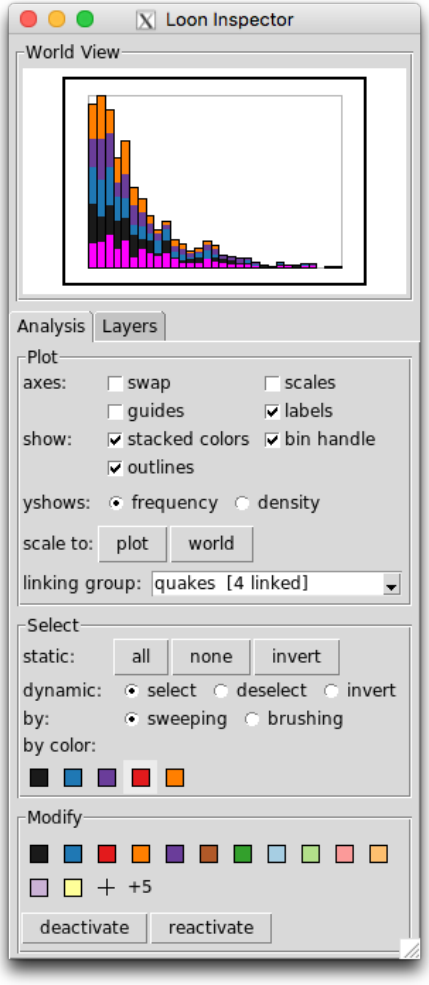
To see where the various groups fall in the histogram, check "stacked colors" box in the plot options section on the "Analysis" tab:

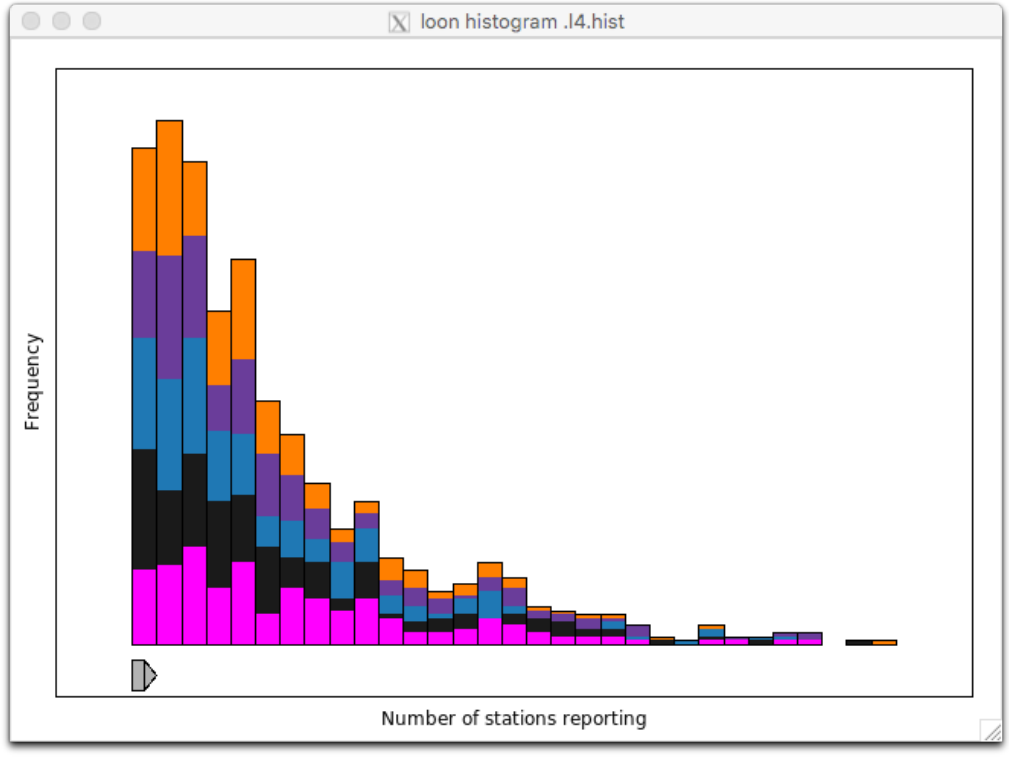




Analysis tab - selection by colour

Selecting by colour will drop the selected colours to the bottom of the histogram.





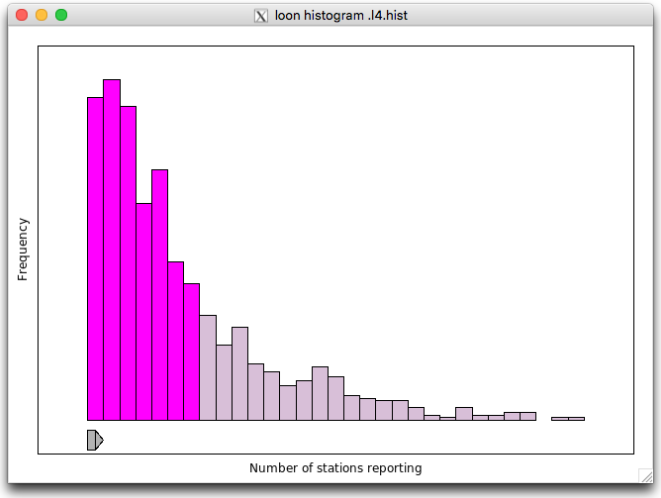
Multiple selection of colours is possible (i.e. via "shift + select") from the inspector, or directly on the histogram itself.

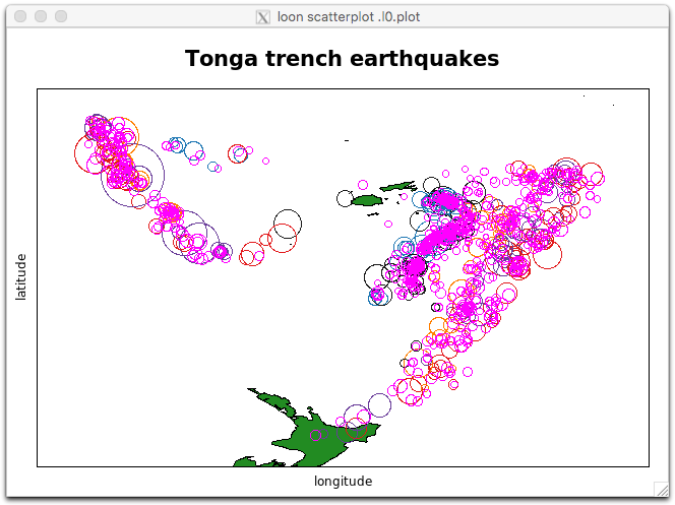
In the latter case, the colours are selected within each bar.

Histogram - selection by colour

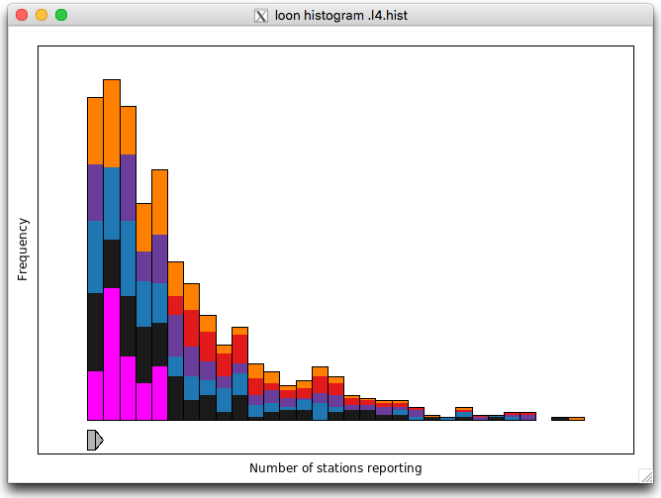
Bars (and colours when stacked) can be selected on the histogram itself:

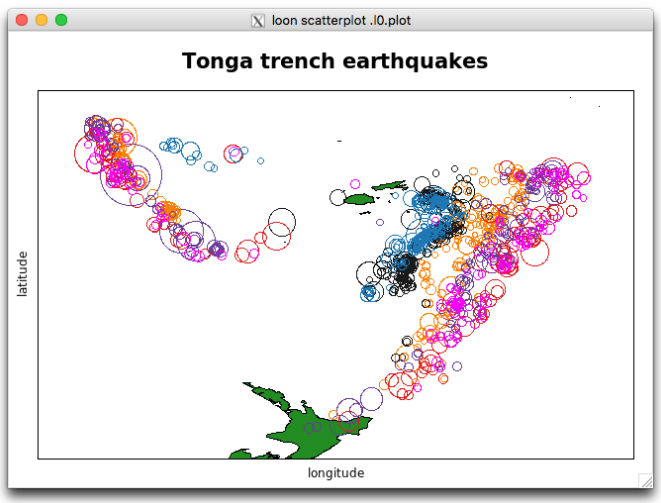
Selecting bars





Selecting colours (a few reds)





Three dimensional scatterplots –

The quakes data has three spatial variables `lat`, `long`, and `depth`, which would be natural to view in a three dimensional plot.

First, because these are three very different scales (degrees latitude and longitude, and kilometres deep) the data needs to be rescaled. The function `l_scale3D()` rescales the variables of its argument to a more nearly common scale.

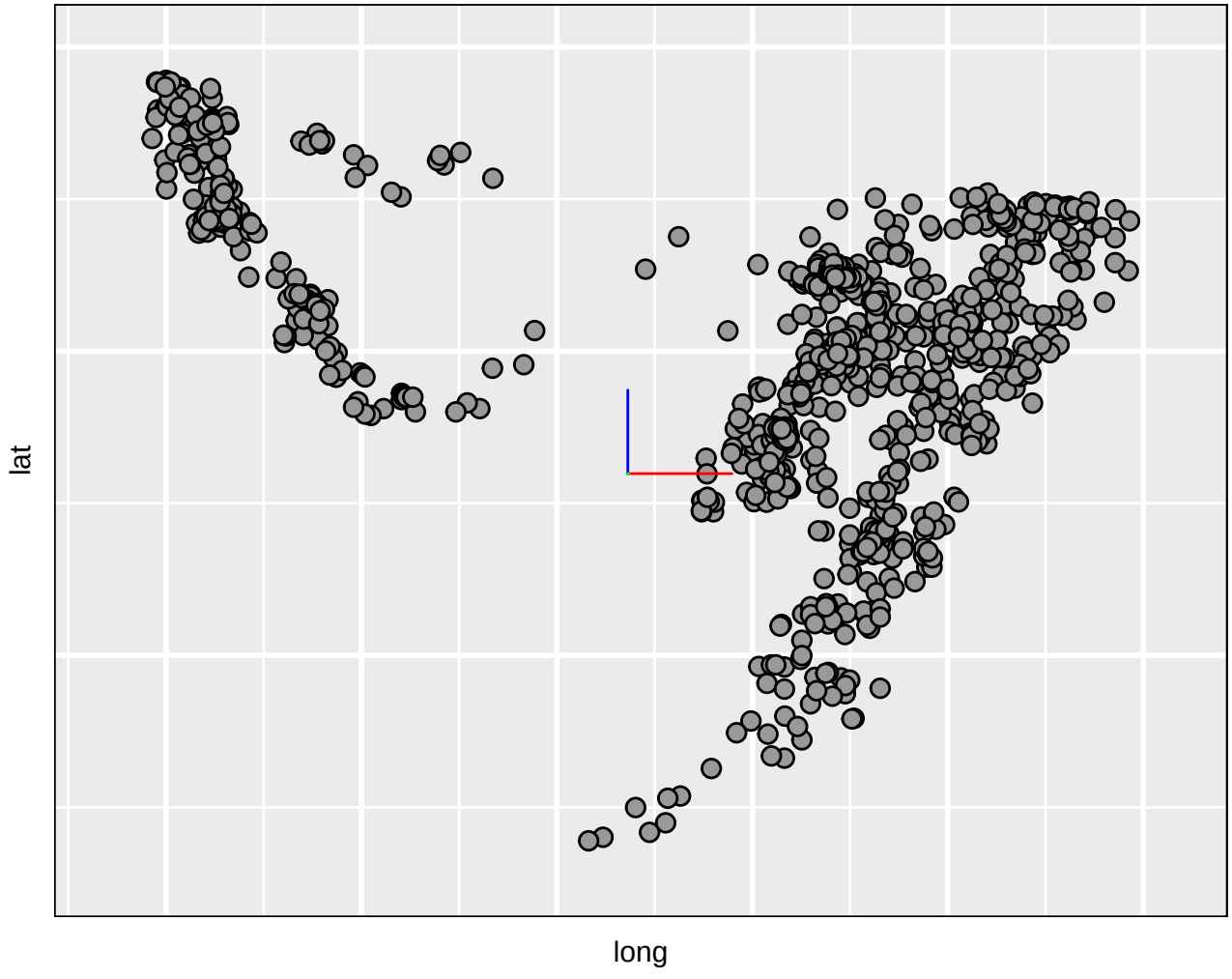
The rescaled data can then be passed to `l_plot3D()` to produce a three dimensional scatterplot.

```
p3D <- with(l_scale3D(quakes),
            l_plot3D(x = long, y = lat, z = depth)
            )
#  Coulr make axis bigger with
#  p3D["axisScaleFactor"] <- 4
```

Three dimensional scatterplots –

The 3D plot looks much like the 2D scatterplot but with an axis shown in colour.

```
plot(p3D)
```



The R key **toggles** rotation mode on and off. Arrow keys rotate as does the left mouse. Right mouse changes the origin of rotation.

Printing a plot - wysiwig or not?

Because `loon` plots are constructed in `tcltk` and their static versions in `grid`, there may be small discrepancies between the two versions (typically in size and font determinations) caused by translations from one system to the other.

These might could be addressed in at least three ways:

1. Adjust the `loon` graphic (e.g. changing glyph sizes) to effect the desired change in the consequent `grid` graphic.
2. Adjust the `grid` object (or `grob`) itself. The translation (and drawing) is carried out by the `loon` function `grid.loon()`, as in

   ```
   gp <- grid.loon(p, draw = FALSE)
   gp
   ```

   ```
   ## gTree[GRID.gTree.66]
   ```

   which produces a `grid` graphics `gtree` containing all the information needed to plot (and change) the translated `loon` plot in `grid`.
3. Most (all?) modern operating systems have a "screen shot" capability that allows (at least) a pixmap image of any window to be captured. This could be used on any `loon` window (including the inspector) and in fact was used to produce most of the `loon` images in this document.

Printing a plot - more on the `grid` connection

Graphics in the `grid` package are built up from graphical objects or `grobs`.

In `loon`, in addition to simply `plot(p)`, functions `grid.loon()` and `loonGrob()` can be used to construct `grobs` from `loon` plots that can in turn be used as any other `grob` in `grid`.

- `grid.loon(p)` translates, draws, and (invisibly) returns the `grob` corresponding the current state of the loon plot `p`.
- `loonGrob(p)` translates and returns the `grob`
- `plot(p)` translates and draws the `grob`

The resulting grob, can be used with all the rich functionality of the `grid` package. For example,

```
library(grid)
grid.ls(gp)  # lists the contents of the grob
# Or, in RStudio, can be viewed interactively
View(gp)
```

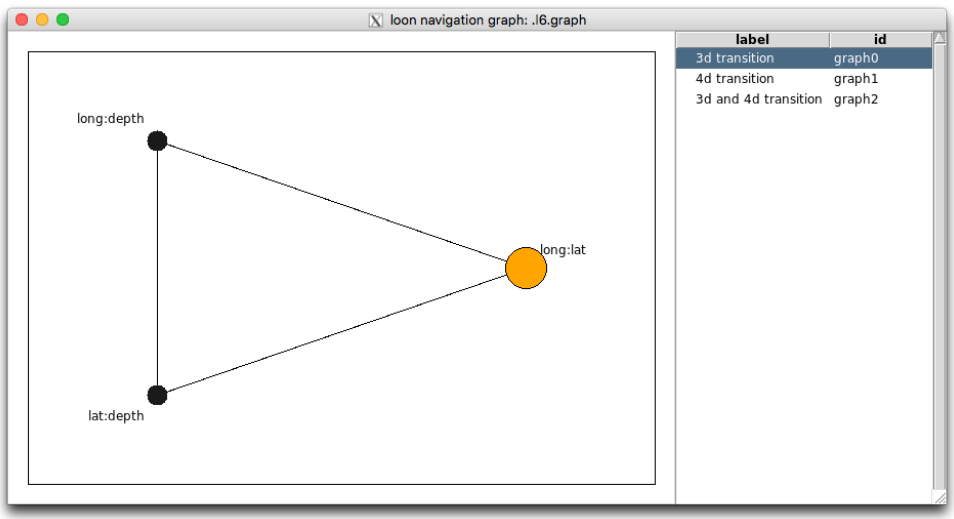
Note that all of the elements of a loon plot appear in the grob, either explicitly or, if they were not drawn in the loon plot, as an empty `grob` containing the arguments relevant to drawing them.

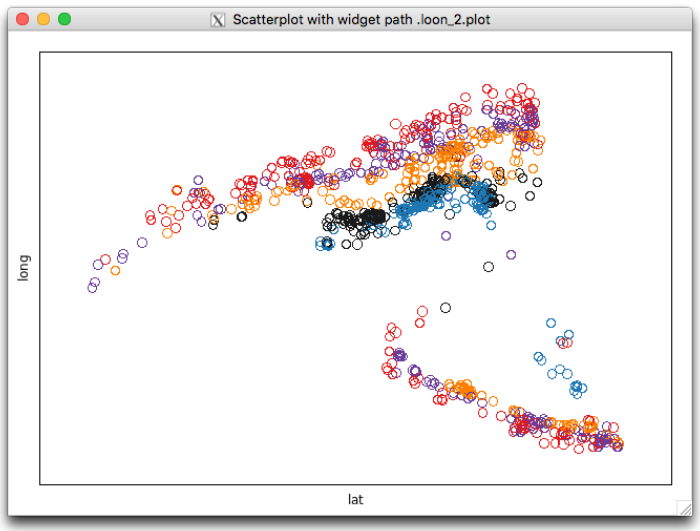
Being able to get and set the plot contents means that it is possible to capture different plot states as `grobs`.

Navigation graphs

`loon` also provides a simple means to examine three (and higher) dimensional structure interactively via its concept of a "navigation graph":

```
# Get a navigation graph (set all sizes to be the same)
ng <- l_navgraph(quakes[,c("long", "lat", "depth")],
                 linkingGroup = "quakes", sync = "pull",
                 glyph = "ocircle", size = 5)
# Note that specifying the size will propagate this throughout
# the linking group OVERRIDING the sync = "pull" argument.
```





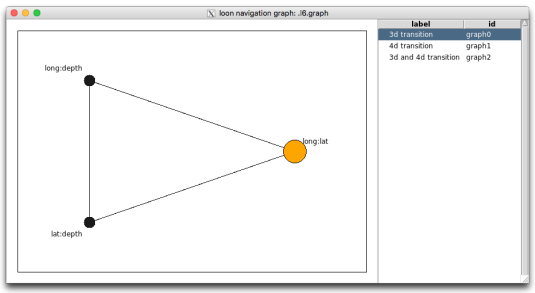
Nodes represent scatterplots of the named variates, edges are 3d transitions found by rotating around the axis of the shared variate.

The large coloured circle, called the "navigator", identifies the scatterplot shown in the separate display at right.

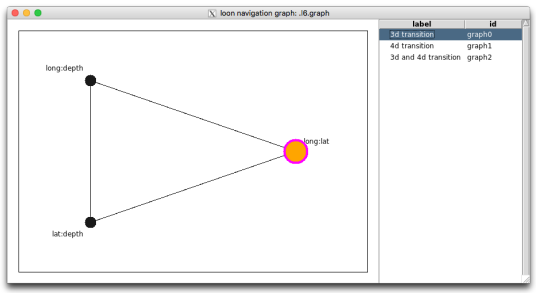
Navigation graphs - the graph navigator

Selecting the navigator highlights which nodes are connected to it. Then one is shift-selected to identify which scatterplot is to appear next.

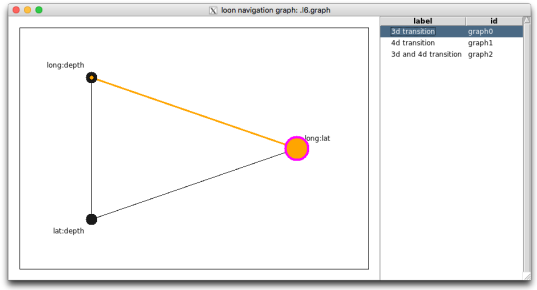
Navigator not selected.



Navigator selected.



Destination node selected.



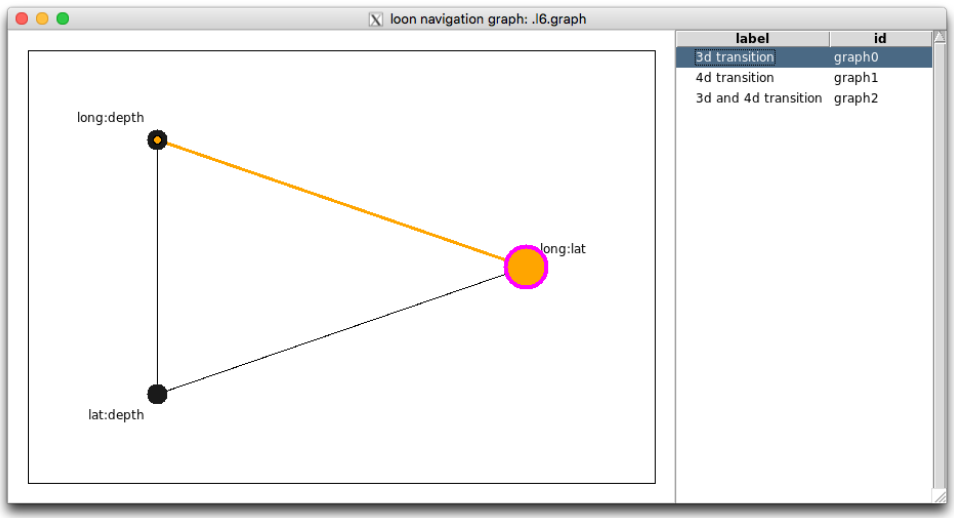
This can continue throughout the graph to produce a path through the three dimensional space.

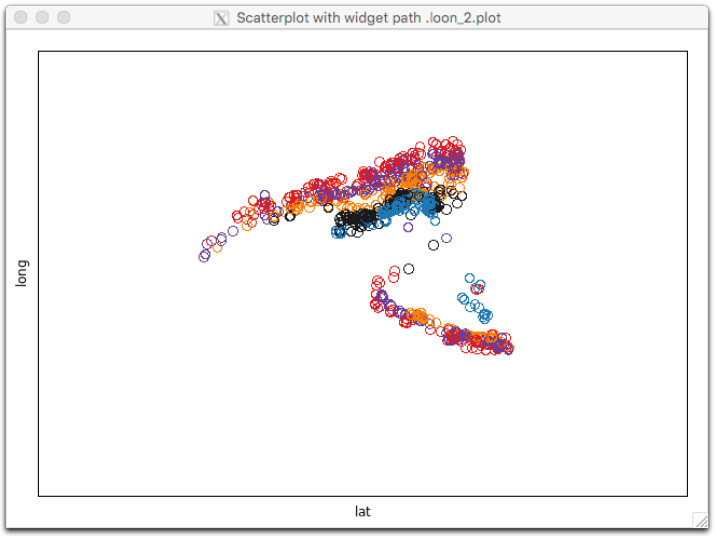
- Connected nodes will highlight at each step.
- The navigator needs to be moved off any node that is to be selected next.
- The navigator can be dragged or moved using scrolling.
- Each edge represents a 3d transition from one 2d space (scatterplot) to the next.

UNIVERSITY OF WATERLOO

Navigation graphs - navigating along a path

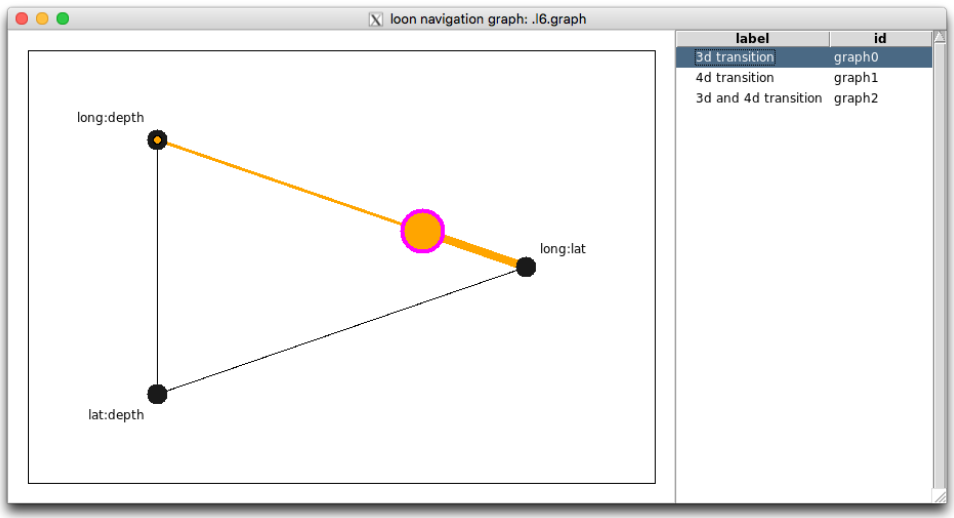
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

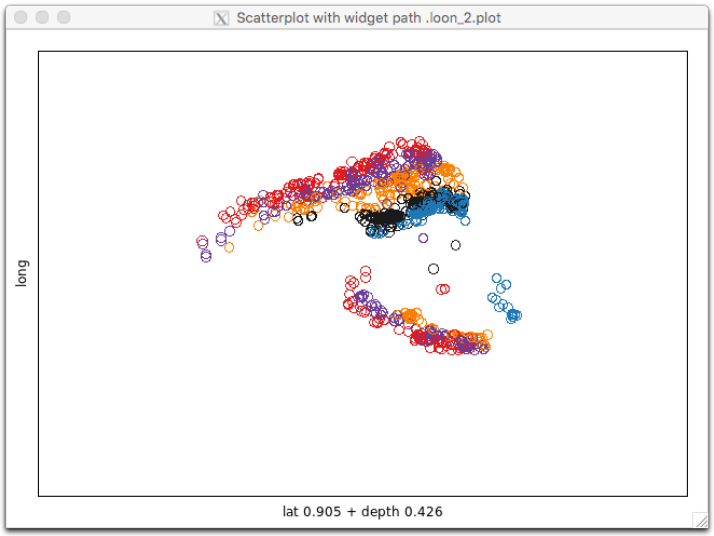




Navigation graphs - navigating along a path

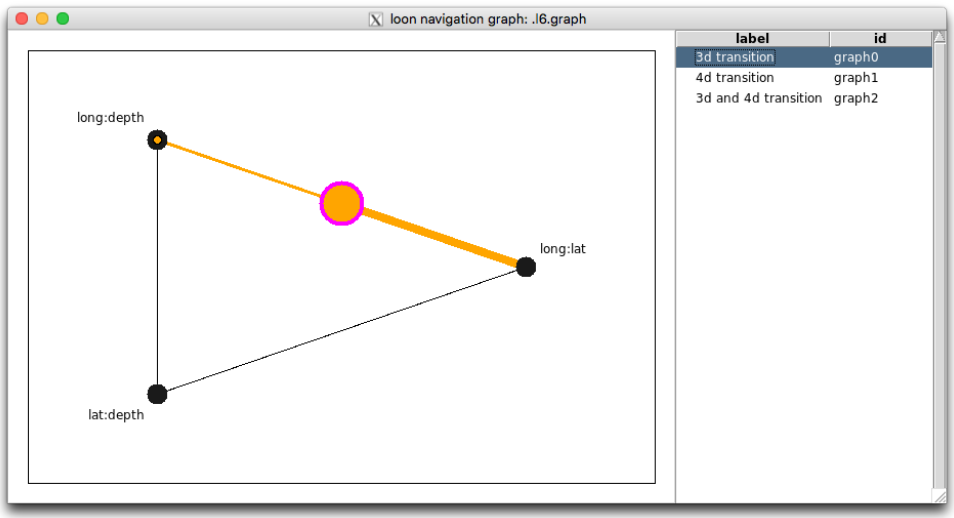
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

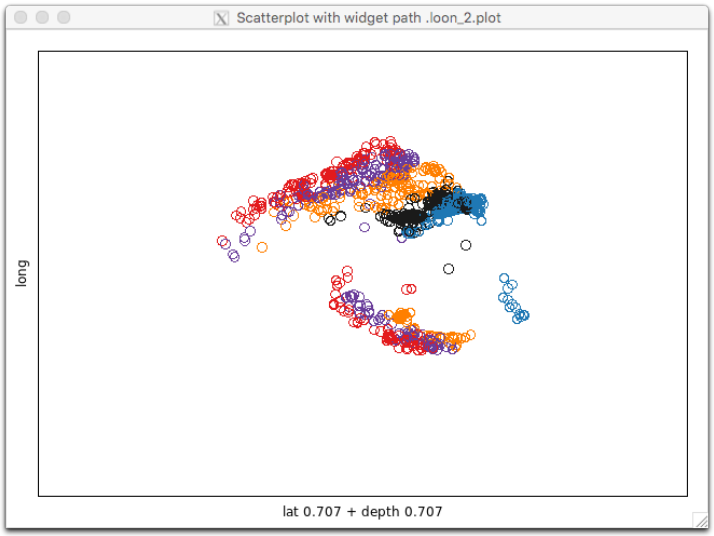




Navigation graphs - navigating along a path

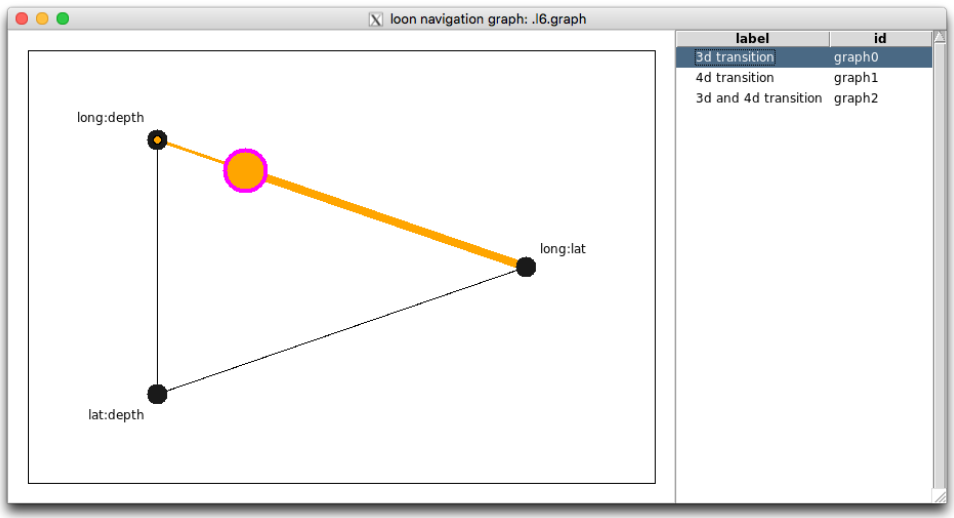
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

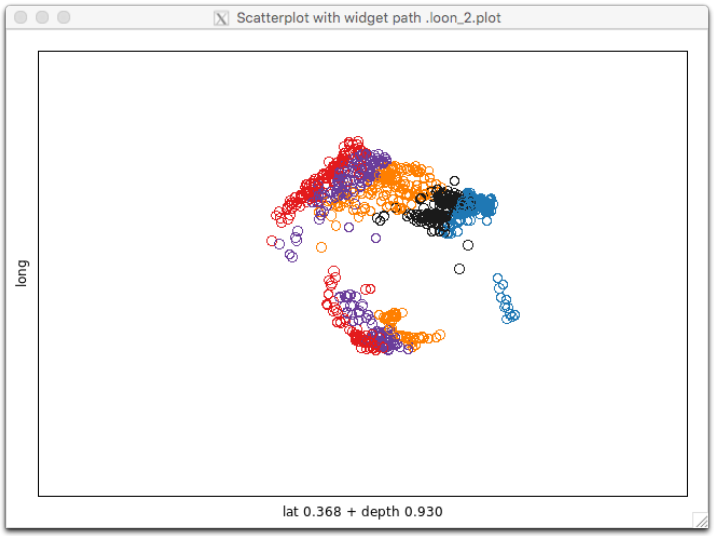




Navigation graphs - navigating along a path

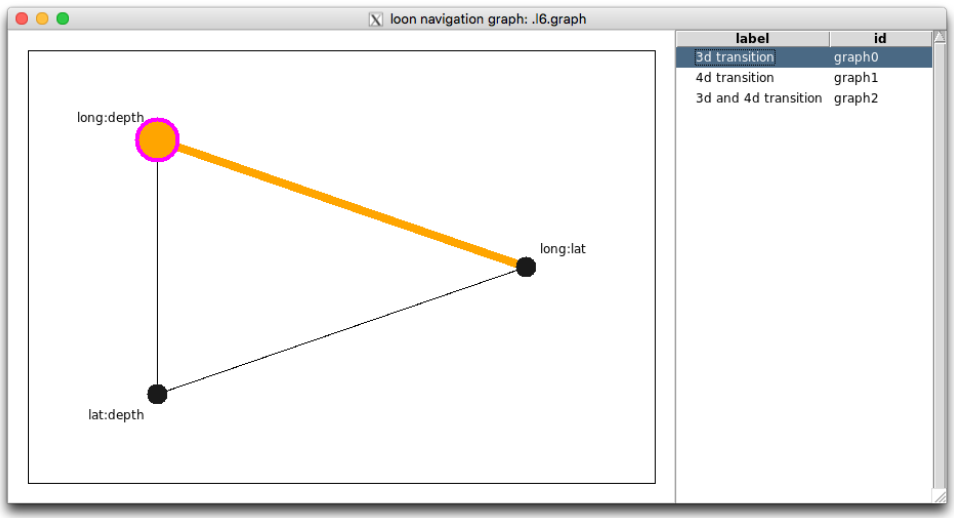
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

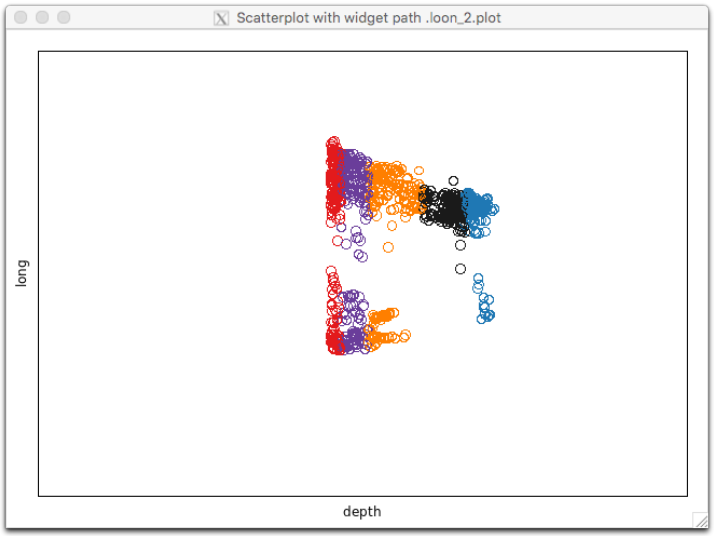




Navigation graphs - navigating along a path

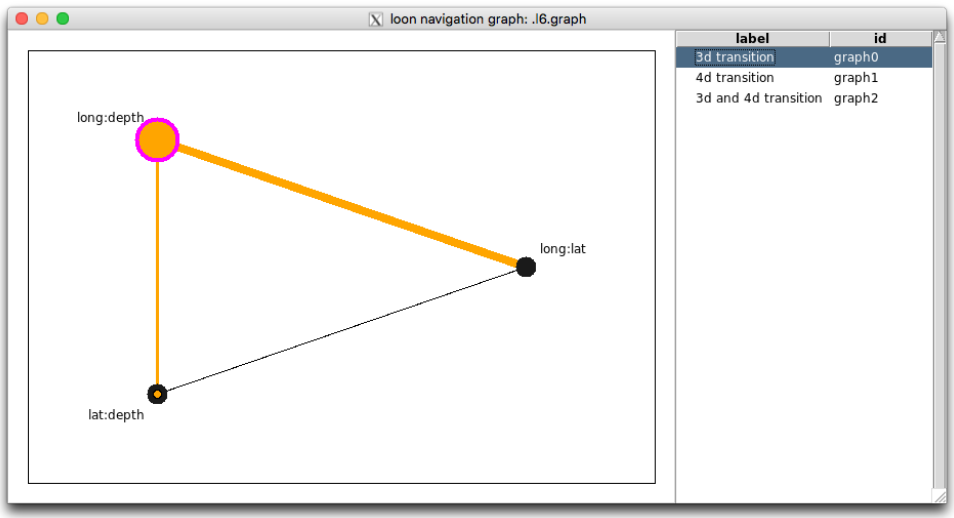
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

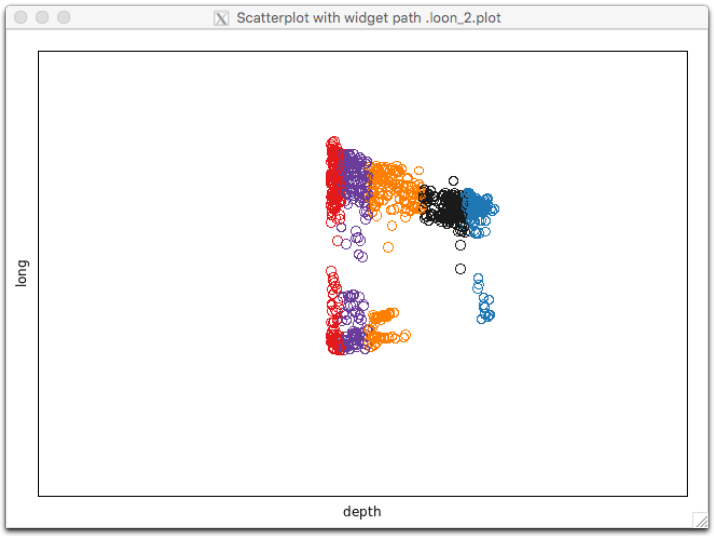




Navigation graphs - navigating along a path

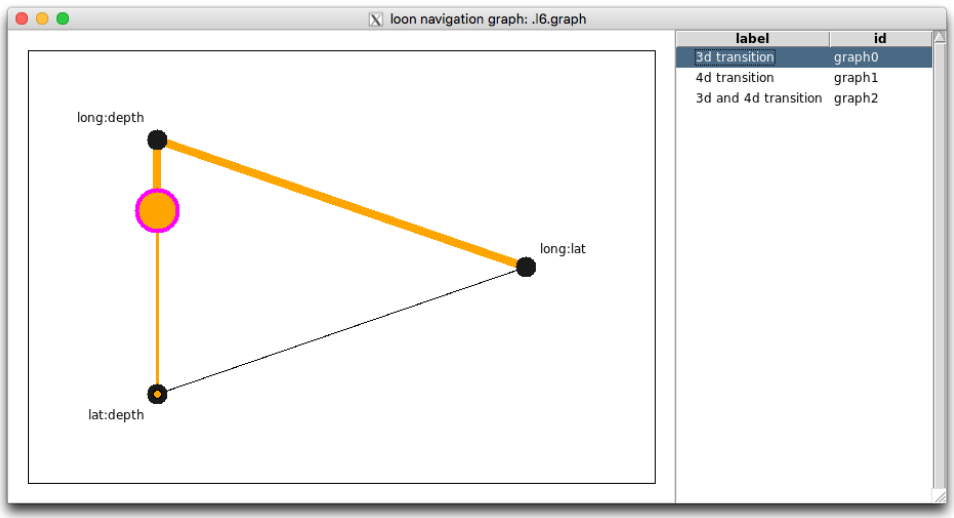
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

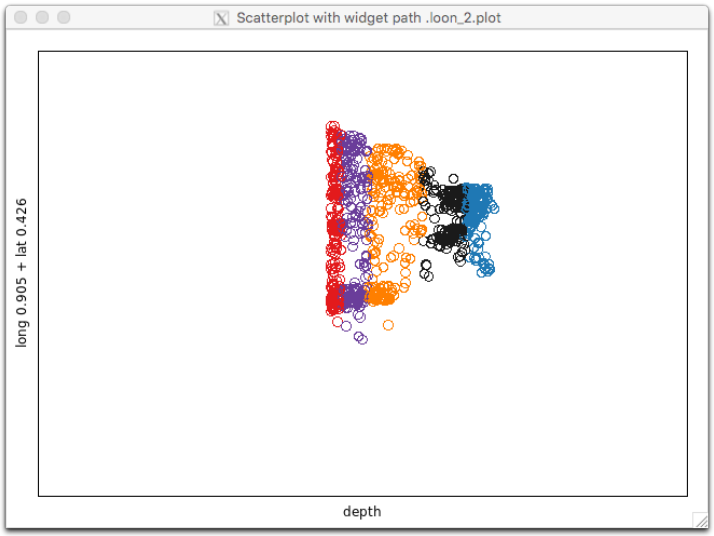




Navigation graphs - navigating along a path

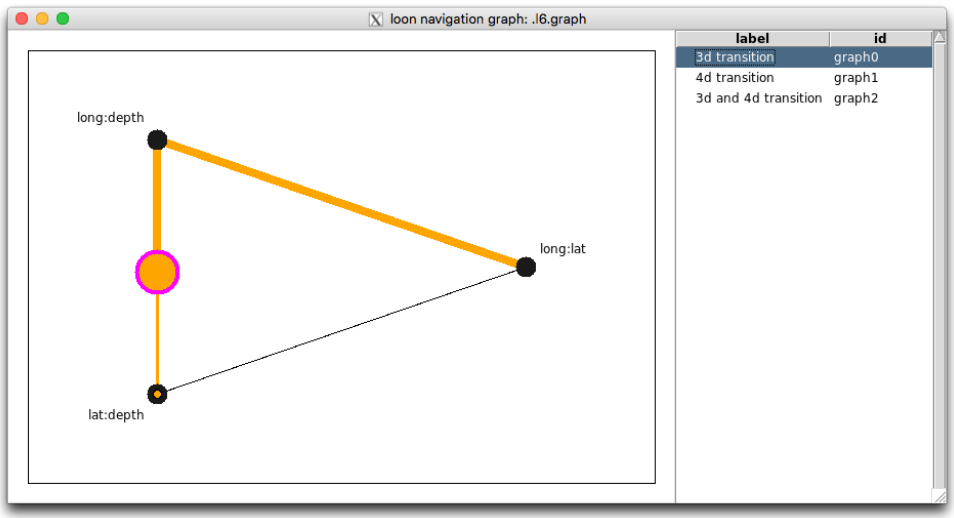
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

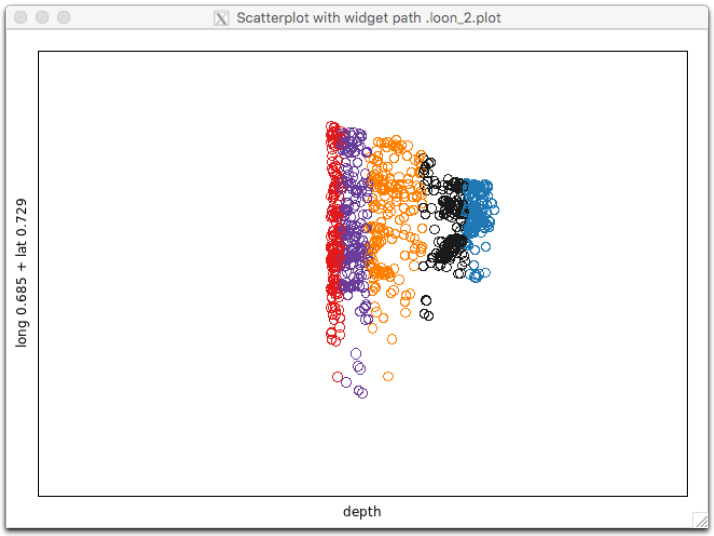




Navigation graphs - navigating along a path

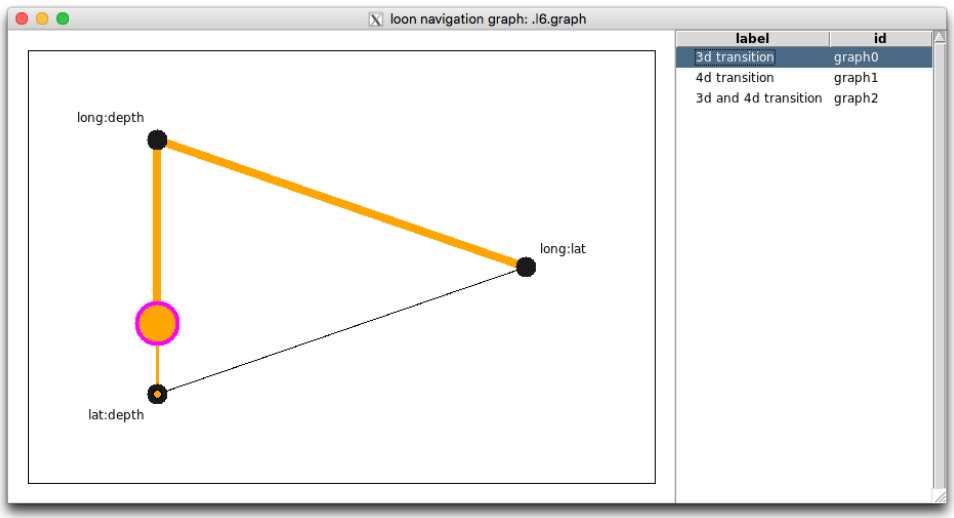
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

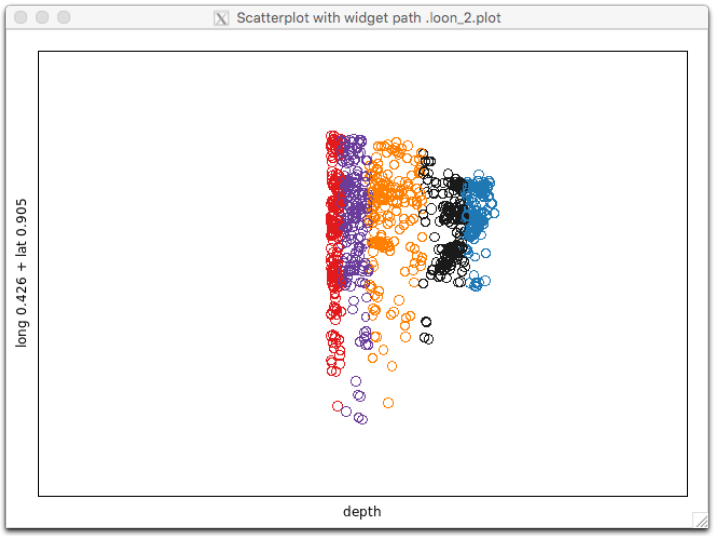




Navigation graphs - navigating along a path

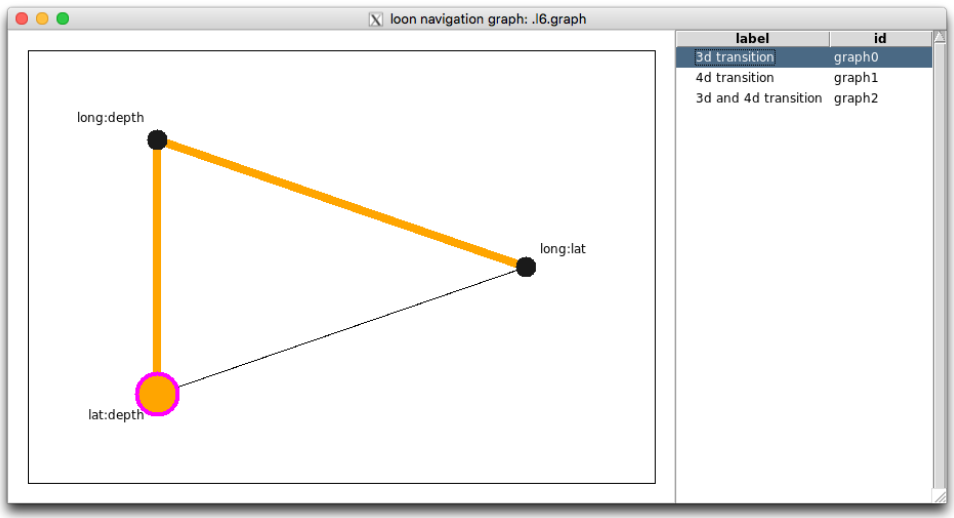
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

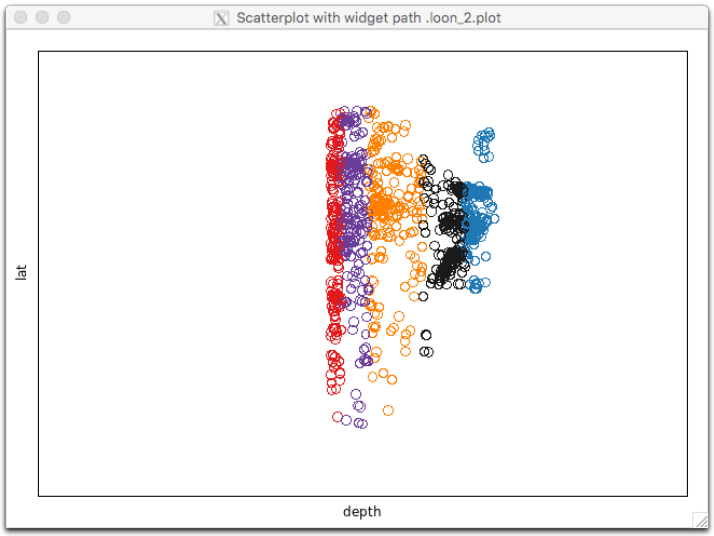




Navigation graphs - navigating along a path

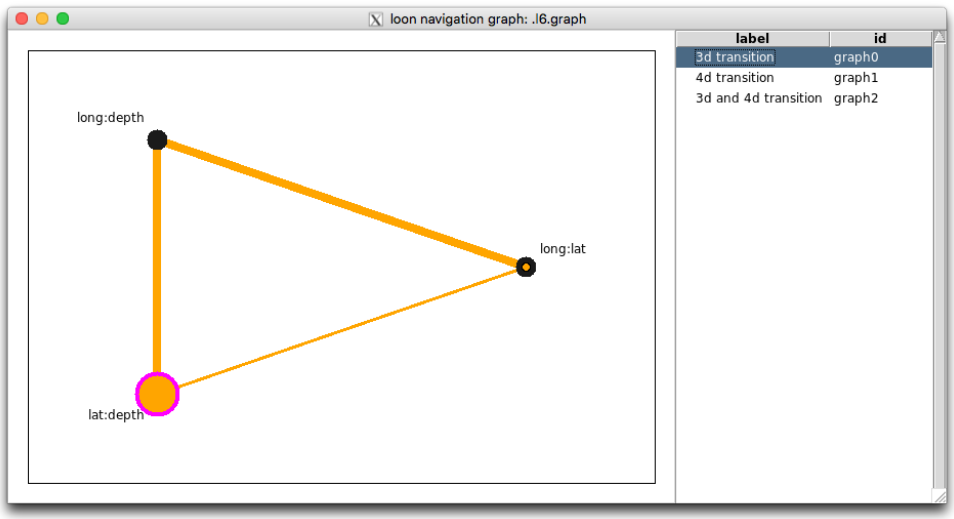
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

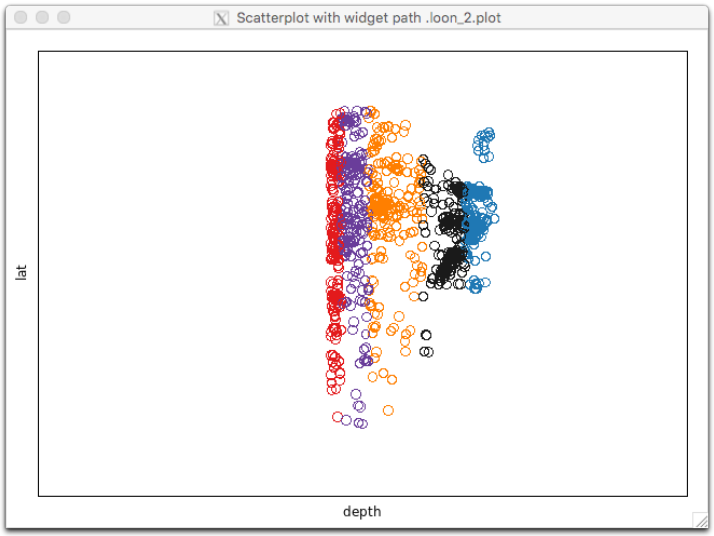




Navigation graphs - navigating along a path

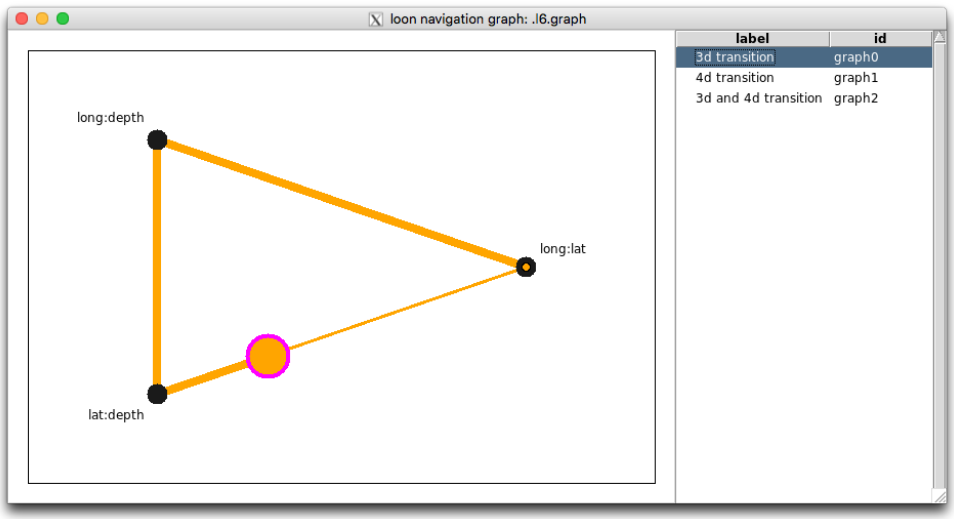
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

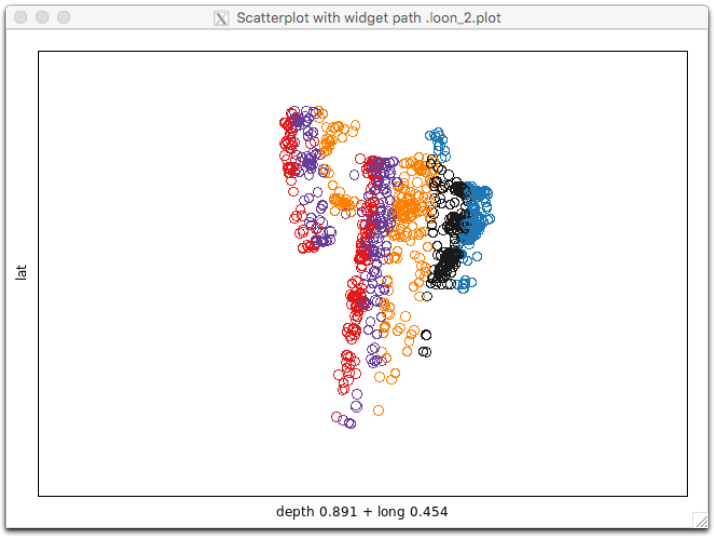




Navigation graphs - navigating along a path

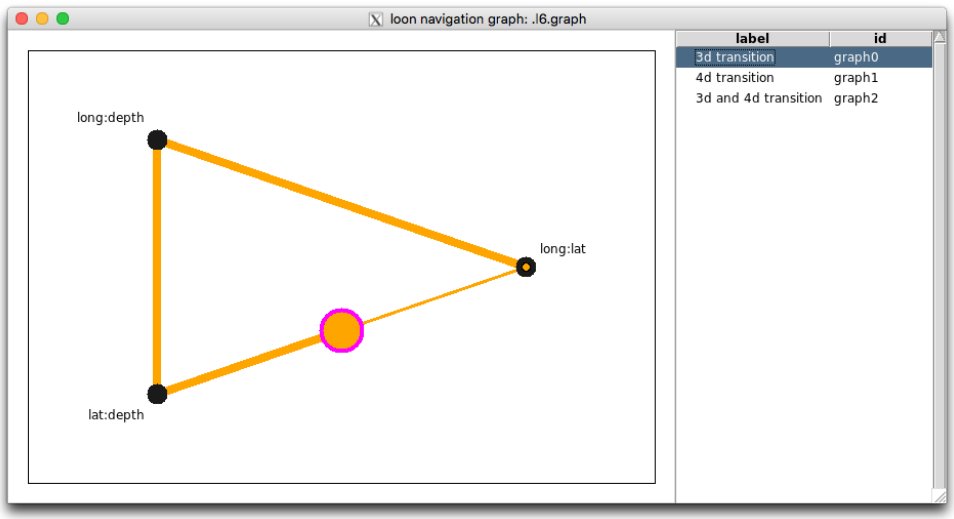
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

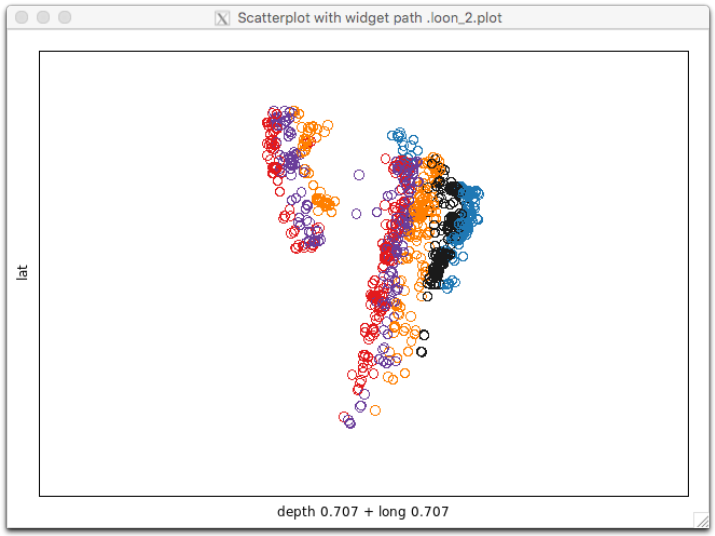




Navigation graphs - navigating along a path

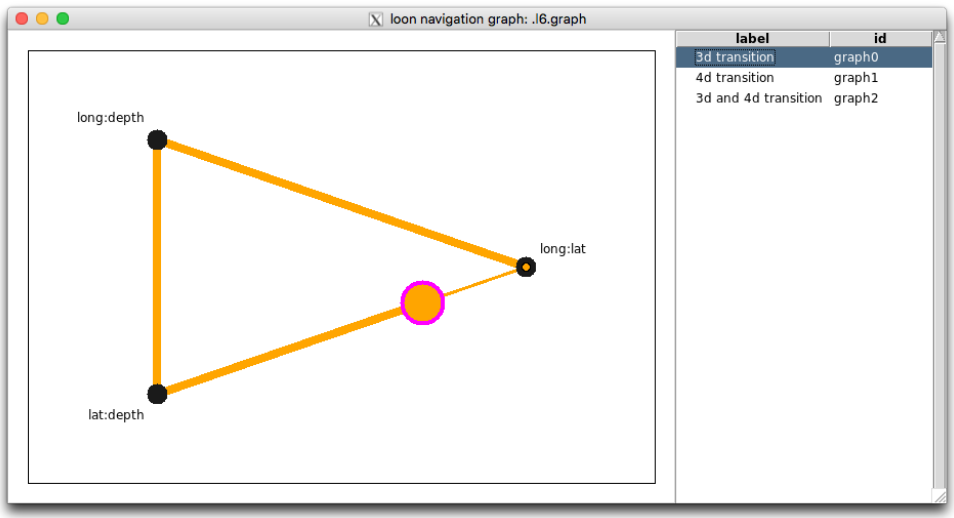
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:

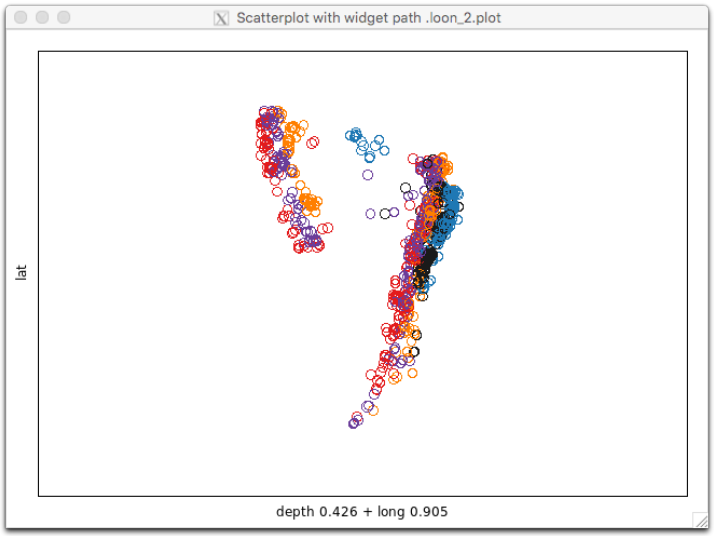




Navigation graphs - navigating along a path

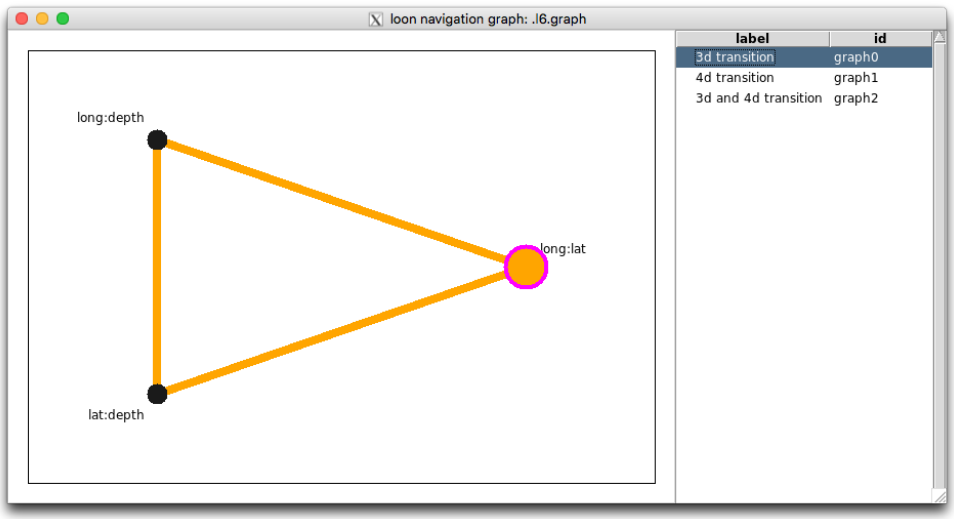
Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:





Navigation graphs - navigating along a path

Moving the navigator along an edge causes the scatterplot to change by rotating the points around the shared axis:



Navigation graph structure - a `list` of class `l_navgraph`

The navigation graph is a list:

```
str(ng)
```

```
## List of 5
##  $ graph      : 'l_graph' Named chr ".l14.graph"
##  $ plot       : 'l_plot' Named chr ".loon_2.plot"
##  $ graphswitch: 'l_graphswitch' Named chr ".l14.graphswitch"
##  $ navigator  : 'l_navigator' Named chr "navigator0"
##   ..- attr(*, "widget")= chr ".l14.graph"
##  $ context    : 'l_context' Named chr "context0"
##   ..- attr(*, "widget")= chr ".l14.graph"
##   ..- attr(*, "navigator")= chr "navigator0"
##   ..- attr(*, "plot")= 'loon' Named chr ".loon_2.plot"
##  - attr(*, "class")= chr [1:3] "l_navgraph" "l_compound" "loon"
```

Note that the list is of class `l_navgraph` as well as `l_compound` and `loon`. Each element is also a loon object of various classes, including the class `loon`.

For example, programmatic access to the scatterplot can therefore be had via `ng$plot`; similarly for the other elements.

Navigation graph structure - as an `l_compound` object

The navigation graph is an 'l_compound:

Because an `l_navgraph` is also an `l_compound`, the generic function `l_getPlots()` will return all elements which are also loon plots of some kind.

This helper function returns a list of plots with names that should be meaningful for that kind of `l_compound`. For example,

```
l_getPlots(ng)
```

```
## $graph
## [1] ".l4.graph"
## attr(,"class")
## [1] "l_graph" "loon"
##
## $plot
## [1] ".loon_2.plot"
## attr(,"class")
## [1] "l_plot" "loon"
```

Navigation graph structure - the graph and the plot

The graph itself is a special kind of `loon` plot. It has all of the `info_states` of `plot` plus several more related to being a graph. It is in fact a subclass of plot (in `tcl` not in `R`).

For example,

```
g <- ng$graph
names(g)
```

```
##  [1] "itemLabel"        "showItemLabels"   "glyph"            "linkingGroup"
##  [5] "linkingKey"       "zoomX"            "zoomY"            "panX"
##  [9] "panY"             "deltaX"           "deltaY"           "xlabel"
## [13] "ylabel"           "title"            "showLabels"       "showScales"
## [17] "swapAxes"         "showGuides"       "background"       "foreground"
## [21] "guidesBackground" "guidelines"       "minimumMargins"   "labelMargins"
## [25] "scalesMargins"    "x"                "y"                "xTemp"
## [29] "yTemp"            "color"            "selected"         "active"
## [33] "size"             "tag"              "useLoonInspector" "selectBy"
## [37] "selectionLogic"   "activeNavigator"  "nodes"            "from"
## [41] "to"               "isDirected"       "activeEdge"       "colorEdge"
## [45] "orbitDistance"    "orbitAngle"       "showOrbit"
```

```
g["nodes"]
```

```
## [1] "long:lat"   "long:depth" "lat:depth"
```

Generally, programmatic access to the scatterplot via `ng$plot` will be of most interest. Again, both are accessible via `l_getPlots(ng)`

UNIVERSITY OF WATERLOO

Navigation graph structure - other elements

The remaining components of an `l_navgraph` are generally only of interest in building new displays. These are the `graphswitch`, the `navigator`, and the `context`.

For example, the `navigator` contains information on its display and interaction:

```
nav <- ng$navigator
class(nav)
```

```
## [1] "l_navigator" "loon"
```

```
names(nav)
```

```
## [1] "tag"                          "color"
## [3] "animationPause"               "animationProportionIncrement"
## [5] "scrollProportionIncrement"    "from"
## [7] "to"                           "proportion"
## [9] "label"
```

Visualizing several dimensions - `l_pairs()`

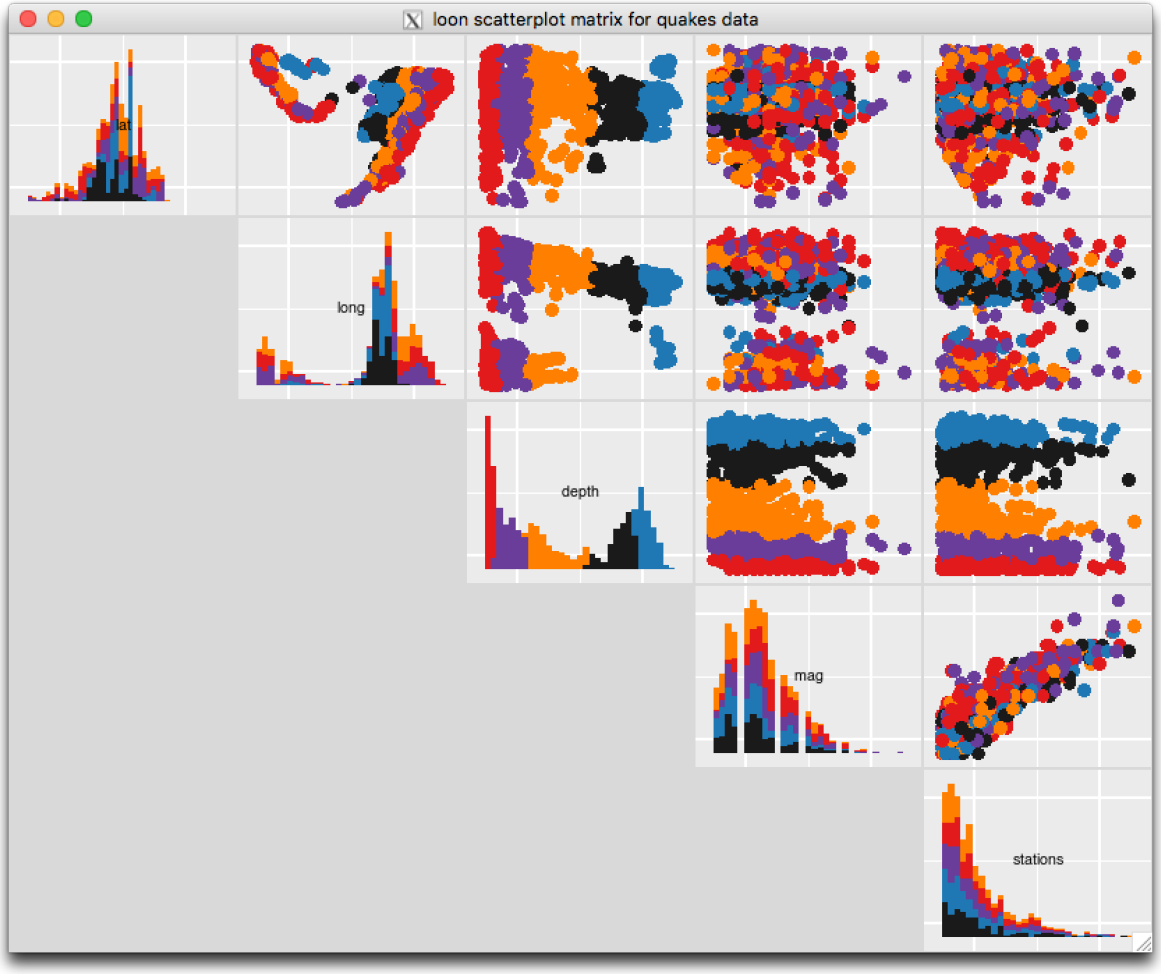
`loon` also provides a simple scatterplot matrix plot analogous to that produced by the base graphics `pairs()`:

```
allPairs <- l_pairs(quakes, showHistograms = TRUE, histLocation = "diag",
                    linkingGroup = "quakes", showGuides = TRUE)
```

Visualizing several dimensions - `l_pairs()`

`loon` also provides a simple scatterplot matrix plot analogous to that produced by the base graphics `pairs()`:

```
allPairs <- l_pairs(quakes, showHistograms = TRUE, histLocation = "diag",
                    linkingGroup = "quakes", showGuides = TRUE)
```



- Each plot is a 'loon' scatterplot and hence can be interacted with as before.
- Panning and zooming of any plot causes the other plots in the same row and/or column to change accordingly

Visualizing several dimensions - `l_pairs()`

`l_pairs()` is a `list` of `loon` plots:

```
str(allPairs)
```

```
## List of 15
##  $ x2y1: 'l_plot' Named chr ".l5.pairs.plot"
##  $ x3y1: 'l_plot' Named chr ".l5.pairs.plot1"
##  $ x4y1: 'l_plot' Named chr ".l5.pairs.plot2"
##  $ x5y1: 'l_plot' Named chr ".l5.pairs.plot3"
##  $ x3y2: 'l_plot' Named chr ".l5.pairs.plot4"
##  $ x4y2: 'l_plot' Named chr ".l5.pairs.plot5"
##  $ x5y2: 'l_plot' Named chr ".l5.pairs.plot6"
##  $ x4y3: 'l_plot' Named chr ".l5.pairs.plot7"
##  $ x5y3: 'l_plot' Named chr ".l5.pairs.plot8"
##  $ x5y4: 'l_plot' Named chr ".l5.pairs.plot9"
##  $ x1y1: 'l_hist' Named chr ".l5.pairs.hist"
##  $ x2y2: 'l_hist' Named chr ".l5.pairs.hist1"
##  $ x3y3: 'l_hist' Named chr ".l5.pairs.hist2"
##  $ x4y4: 'l_hist' Named chr ".l5.pairs.hist3"
##  $ x5y5: 'l_hist' Named chr ".l5.pairs.hist4"
##  - attr(*, "class")= chr [1:3] "l_pairs" "l_compound" "loon"
```

Since `l_pairs` is also an `l_compound`, these could also have been accessed as `l_getPlots(allPairs)`. That way, the user can always rely on `l_getPlots()` to return **only** the plots in the `l_compound`. Again, the names are meaningful in the context of the particular `l_compound`.

UNIVERSITY OF WATERLOO

Visualizing several dimensions - `l_pairs()`

For example, to have every scatterplot in the display also link on the value of its `glyph`, and to turn off the stacking of colours in the histograms:

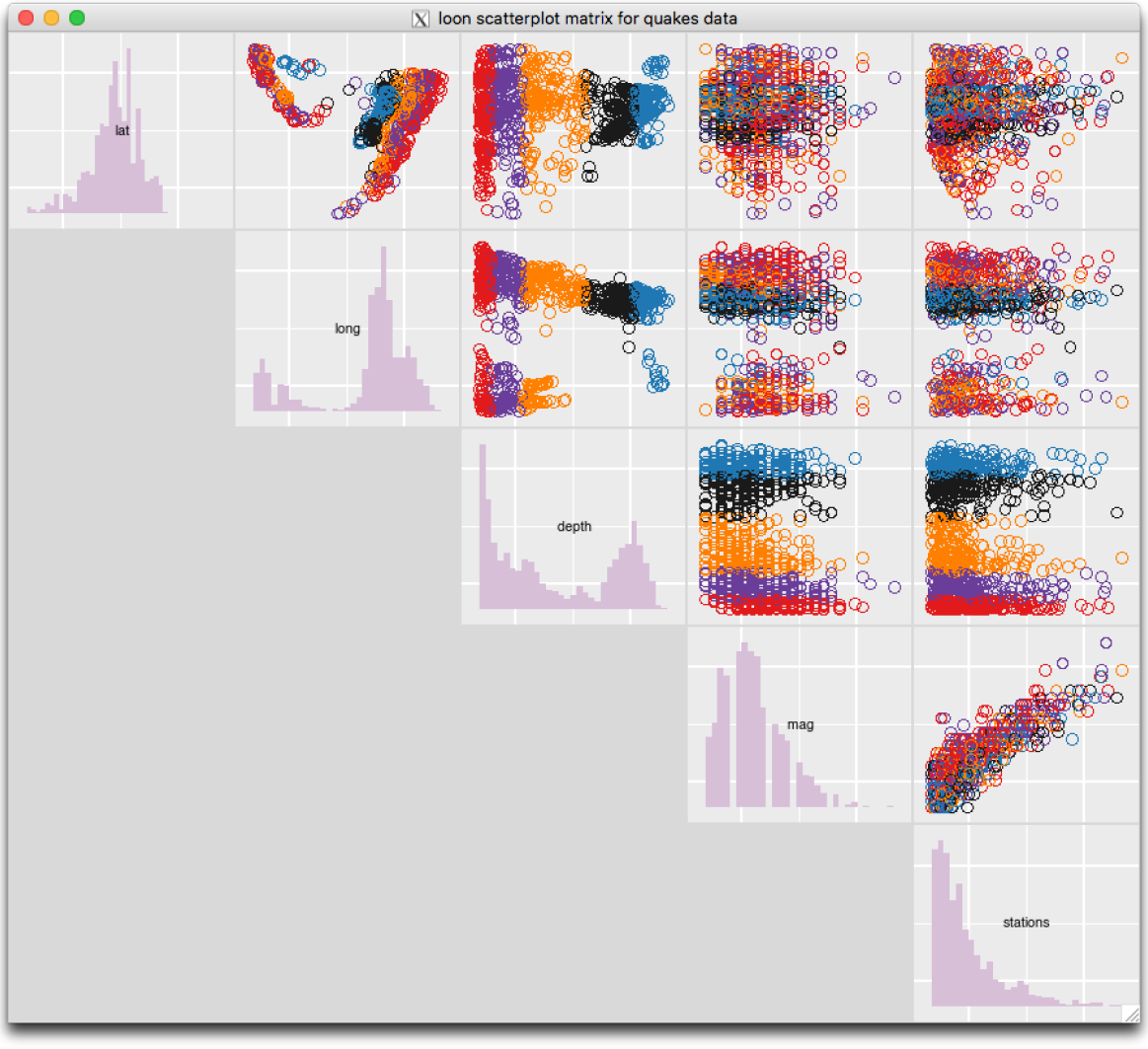
```
for (plot in l_getPlots(allPairs)) {
    if (is(plot, "l_plot")){ # Must be an l_plot
        l_setLinkedStates(plot,
                          c(l_getLinkedStates(plot), "glyph"))
    }
    if (is(plot, "l_hist")){ # Must be an l_hist
        plot["showStackedColors"] <- FALSE
    }
}
```

Note: the linking is not effected until scatterplots in the `linkingGroup` change their `glyph`.

UNIVERSITY OF WATERLOO

Visualizing several dimensions - `l_pairs()`

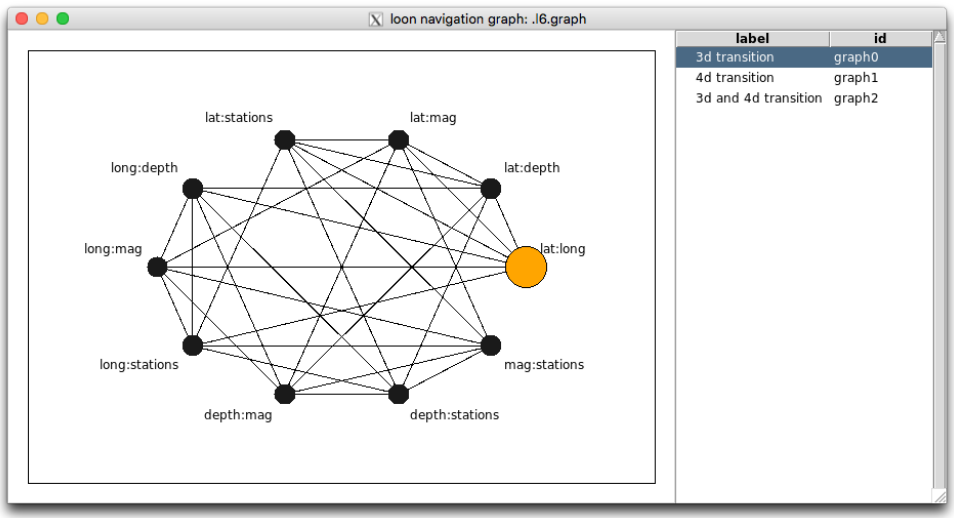
Once `p` (which has also registered `glyph` as a linked state) updates its `glyph`, so too now will `allPairs`:

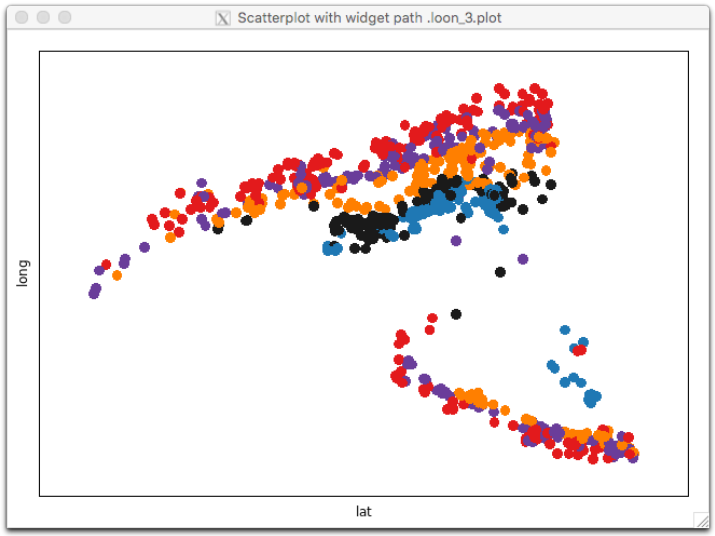


Visualizing several dimensions - `l_navgraph()`

With all six variates, the value of a navigation graph for exploring higher dimensional data can be appreciated:

```
ng_all <- l_navgraph(quakes, linkingGroup = "quakes", sync = "pull")
```

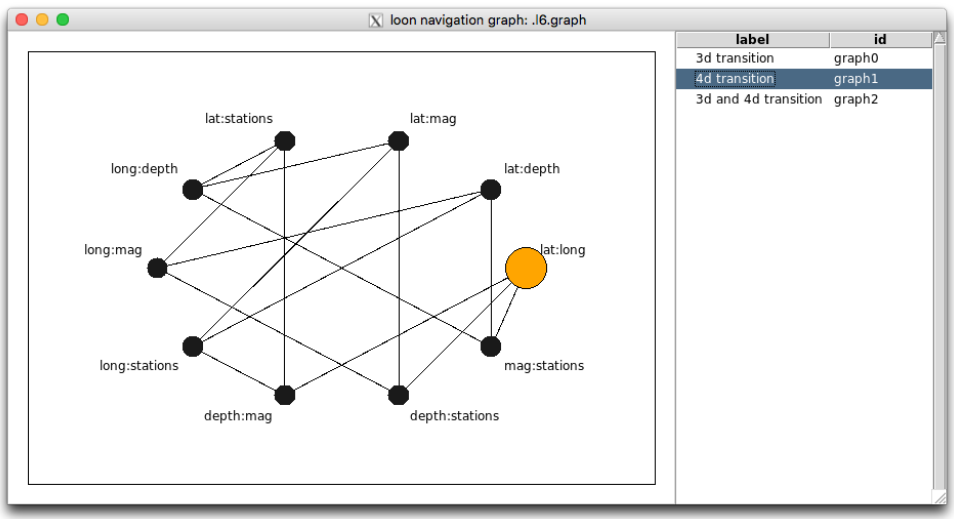


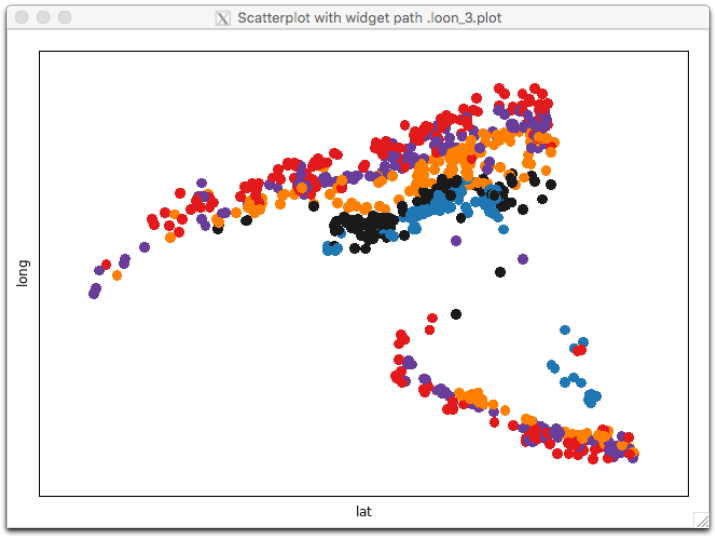


There are now $\binom{5}{2} = 10$ nodes. Edges exist only between nodes which share a variate – a 3d transition graph.

Visualizing several dimensions - `l_navgraph()`

Selecting `4d transition` will produce the complement of the previous graph:





Now edges exist only between nodes which share no variates – a 4d transition graph.

Instead of rigid 3d rotations, now moving the navigator along an edge causes a 2d plane to move along a geodesic through the 4d space from the 2d space of one pair of variates to the 2d space of the others.

Adding information to observations - `itemLabel`

It will often be convenient to have some extra information available in the form of a "tooltip" for each observation.

Often this could simply be the name or number identifying each observation. It could also be something a little more informative, such as a table of values.

For the `quakes` data, for example we might attach the following information to the plot as the value of its `itemLabel`:

```
# Make a little table of info:
indent <- "  "  # might also use tab character \t"
quakeLabels <-  paste0("quake number: ", rownames(quakes), "\n",
                       indent, "magnitude: ", quakes[,"mag"], " Richter", "\n",
                       indent, "longitude: ", quakes[,"long"], "\n",
                       indent, "latitude: ", quakes[,"lat"], "\n",
                       indent, "depth: ", quakes[,"depth"], " km", "\n"
)
```

And we can now attach these to the previous `l_plots` (but not the histograms):

```
plots <- append(list(map = p, scatterplot = p2,
                     nav_3d = ng$plot, nav_5d = ng_all$plot),
                l_getPlots(allPairs))
for (plot in plots) {
    if (is(plot, "l_plot")){
        plot["itemLabel"] <- quakeLabels
        plot["showItemLabels"] <- TRUE
    }
}
```

Visualizing several dimensions - `l_serialaxes()`

One problem with high-dimensional data is that a Cartesian coordinate system is based on rectilinear layouts which require each new axis to be orthogonal to all the others. This becomes impossible to display simultaneously beyond three dimensions.

In a non-Cartesian coordinate system, the axes are not restricted to being mutually orthogonal. They could, for example, be parallel to each other or, alternatively, present at various angles with one another.

This freedom from orthogonality means that many more coordinates can be laid out in a 2d plane (or, in principle, in a 3d space).

Nevertheless, the axes will still have to appear in some order, necessarily emphasizing the relation between neighbouring axes.

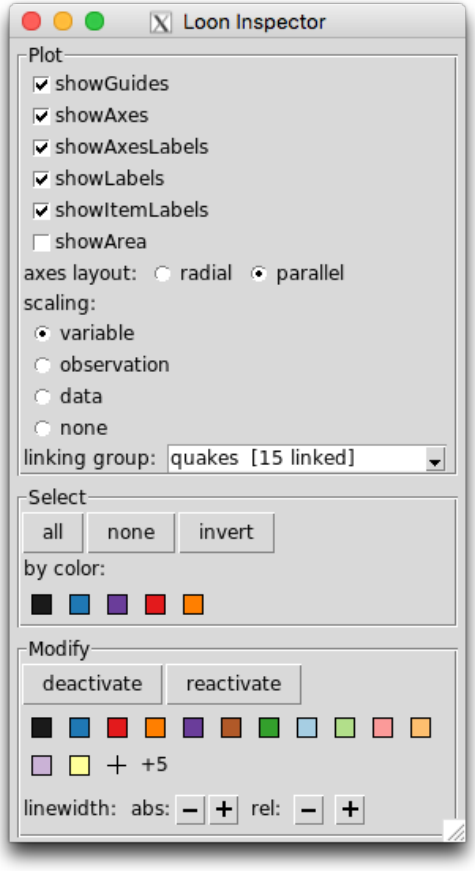
A "serial axes" display is one in which the axes are laid out in series, typically in parallel to one another (parallel coordinates) or arranged as equi-angular radii about some origin (radial coordinates).

The latter are sometimes called star glyphs or star plots (or any of a variety of more colourful names like "radar chart", "spider chart", or "cobweb chart"); as in `loon`, we will simply call them "radial axes" plots as one of two arrangements of serial axes.

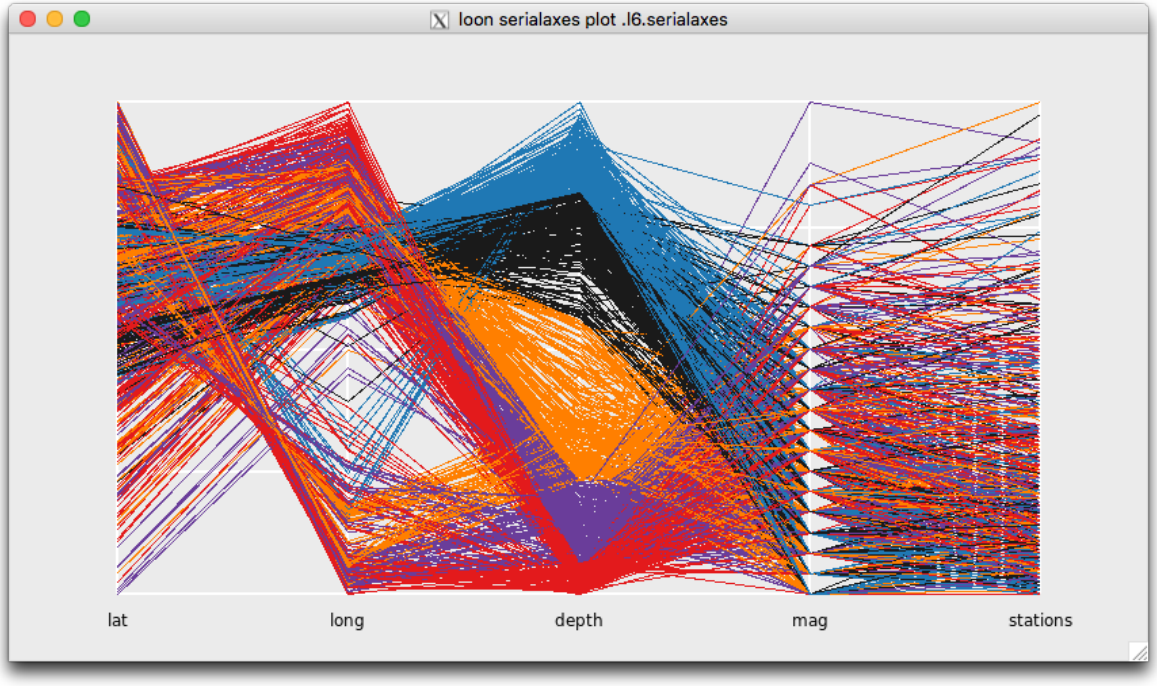
l_serialaxes() - parallel coordinates

```
serialAxes <- l_serialaxes(quakes, axesLayout = "parallel",
                           itemLabel = quakeLabels,
                           showItemLabels = TRUE,
                           linkingGroup = "quakes")
```

Serial axes inspector



Serial axes (parallel coordinate) plot



Each 5-dimensional observation is a "curve" in parallel coordinates.

Note that group selection (sweeping/brushing) does not appear in the inspector. Groups in a serial axes plot are selected using an (always available) extendible line segment.

Parallel coordinates - the line-point duality

Each pair of parallel axes corresponds to a pair of orthogonal axes. Patterns seen in a scatterplot should also appear as patterns in the parallel coordinates.

There is in fact a **line-point duality** that connects the geometry within a orthogonal coordinate pair system to that of the geometry of a parallel coordinate pair system.

- A point in a scatter plot (pair of orthogonal coordinates) is a line between parallel axes (a pair of parallel coordinates)
- A line in a scatter plot (pair of orthogonal coordinates) is a point of intersection of lines between parallel axes (a pair of parallel coordinates)
  - intersection may occur **outside** the parallel axes, possibly at infinity
  - i.e. if you continue the lines beyond the axes they will intersect
- two parallel lines (same slope) in a scatterplot correspond to two different intersection points one above the other in a parallel coordinate plot;
- lines with common intercepts in a scatterplot **do NOT produce** points of intesections in the parallel coordinate plot at the same vertical location; changing slope changes the horizontal **and** vertical position of the point of intersection.
- a curve in a scatterplot is a curve of intersection points in parallel coordinates (think tangent lines, or lines formed by successive points along the curve)

Parallel coordinates - the line-point duality

These points can be easily illustrated using `loon`. Two different files containing illustrative code is available on the course website (use `source("FullPathAndFileName.R")` to load):

- "parallelCoordPointLine.R" illustrates the line-point duality.
- "parallelCoordCorrelation.R" illustrates the relation between parallel coordinates and correlations using several examples.

Loading either file will produce several displays, so you might just load one at a time to avoid clutter.

Once loaded, you will need to interact with each scatterplot as follows:

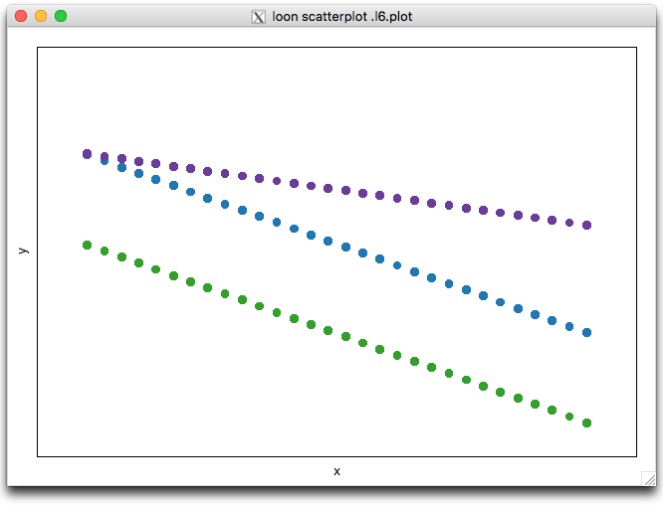
- select any group of points of interest
- from the inspector, invert the others and deactivate them;
- brush points in the scatterplot or lines in the parallel coordinate plot to observe the connections
- select subsets of points in the scatterplot and
  - move them around, or
  - jitter selected points.

Reactivate all points, select and return all to their original position. Repeat with different groups (select by colour).

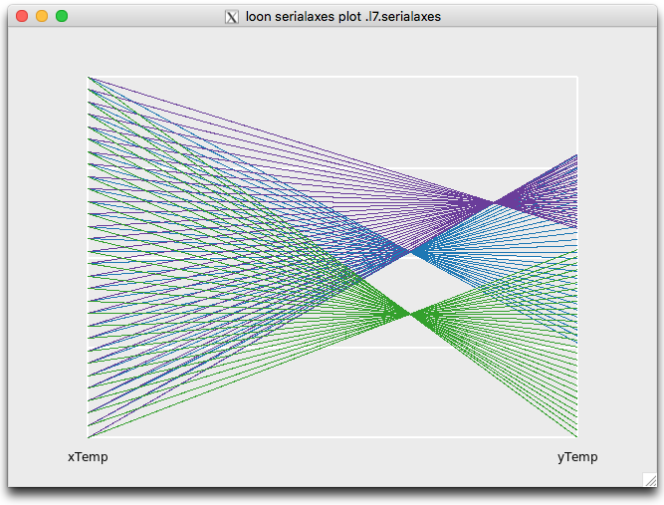
Parallel coordinates - the line-point duality

For example,

Orthogonal coordinates



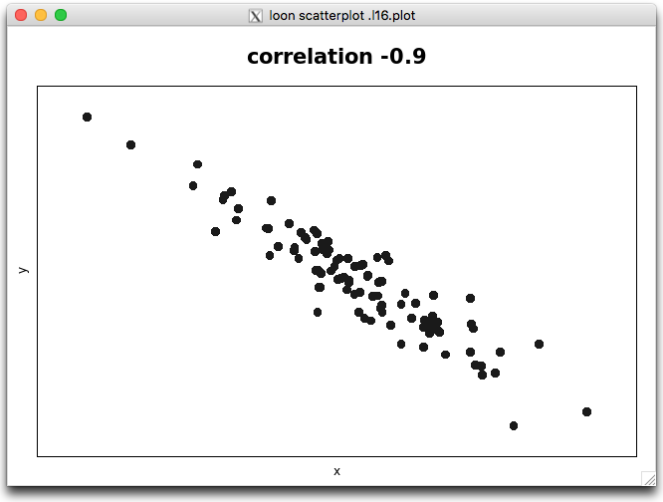
Parallel coordinates

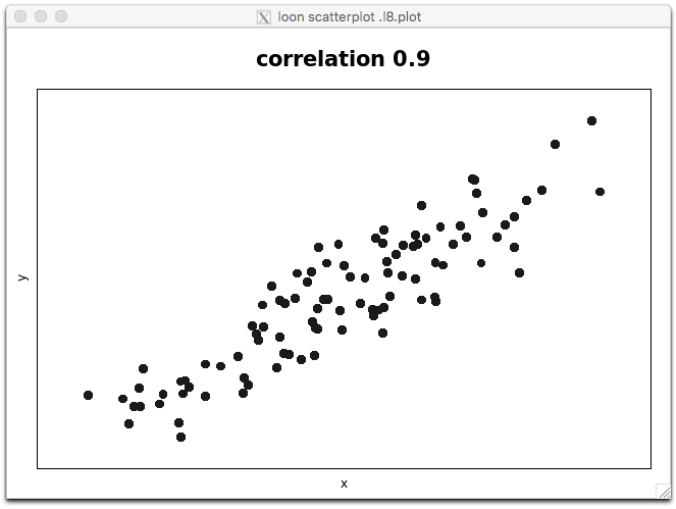


Note the correspondence between parallel lines in the orthogonal coordinates and aligned points in the parallel coordinate space.

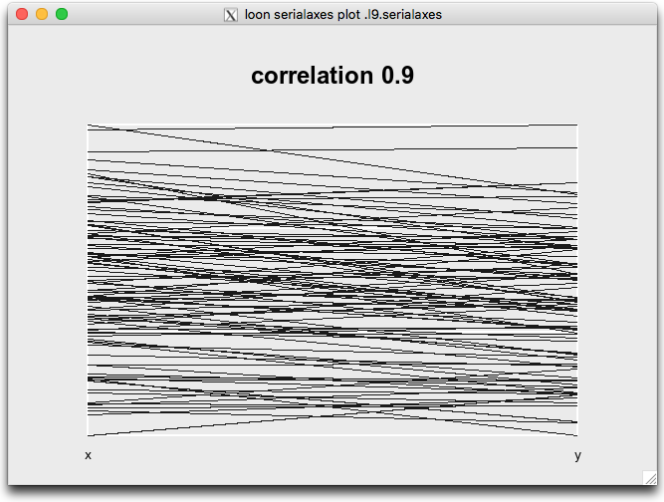
Parallel coordinates - correlation

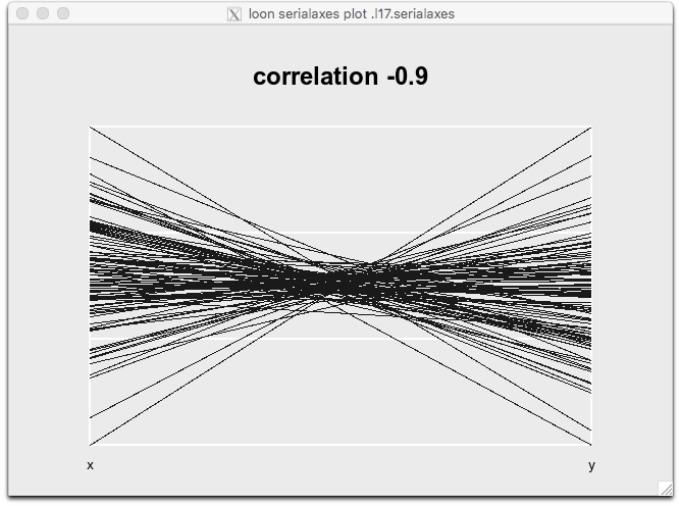
Orthogonal coordinates





Parallel coordinates





l_serialaxes() - states

A serial axes plot also has states, some of which are shared by other loon plots (e.g. `linkingGroup`) and some of which are peculiar to it (e.g. `axesLayout` and `lineWidth`):

```
names(l_info_states(serialAxes))
```

```
##  [1] "linkingGroup"     "linkingKey"       "itemLabel"        "showItemLabels"
##  [5] "showAxes"         "showAxesLabels"   "linewidth"        "scaling"
##  [9] "axesLayout"       "showArea"         "axesLabels"       "data"
## [13] "sequence"         "active"           "color"            "selected"
## [17] "showGuides"       "showLabels"       "useLoonInspector" "title"
## [21] "tag"
```

Many of these are modifiable from the loon inspector; all of them can be modified programmatically.

Two of particular interest are `sequence` and `scaling` affecting the ordering of the axes and the scaling of the observations to fit on the axes.

l_serialaxes() - the sequence state

sequence contains the variates in the order in which they appear on the axes.

```
serialAxes["sequence"]
```

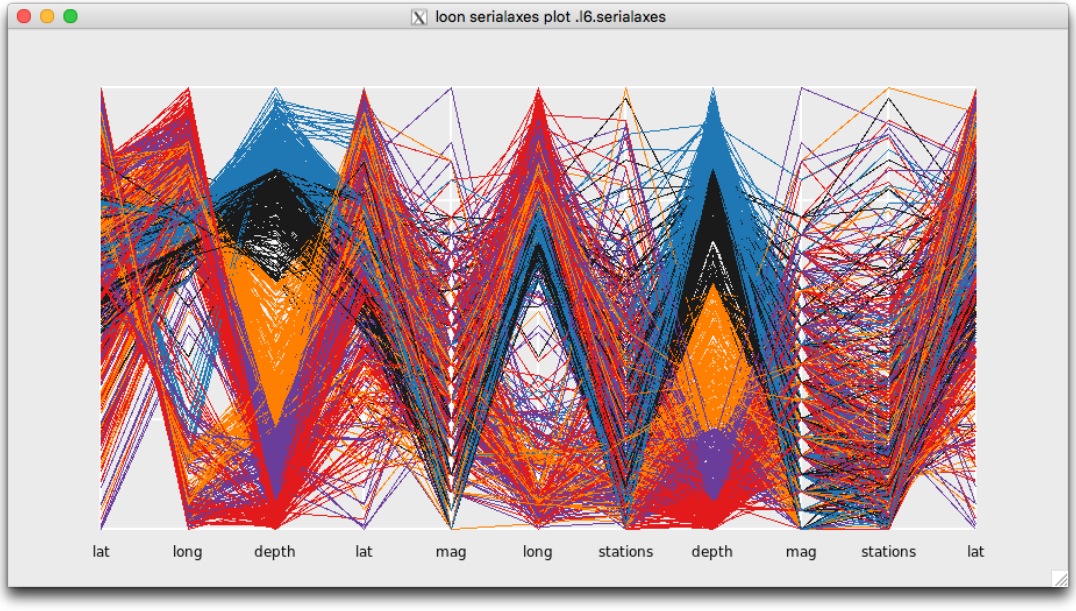
```
## [1] "lat"      "long"     "depth"    "mag"      "stations"
```

Note that axes can be repeated. For example,

```
serialAxes["sequence"] <- serialAxes["sequence"] [c(1, 2, 3, 1, 4, 2, 5, 3, 4, 5, 1)]
serialAxes["sequence"]
```

```
##  [1] "lat"      "long"     "depth"    "lat"      "mag"      "long"
##  [7] "stations" "depth"    "mag"      "stations" "lat"
```

ensures that every variate will appear next to every other variate in the display.



- This allows the relationship between every pair of variates to be seen and compared
- 'eulerian()' (from the `PairViz`) package) returns a (numerical) sequence in which every pair of variates are adjacent at least once.
- other ordering methods are available from 'PairViz'

l_serialaxes() - the scaling state

For plotting purposes, each axis ranges from 0 to 1. Unless the observations themselves have values between 0 1nd 1, they will have to be rescaled before being located on the serial axes.

The `scaling` determines the set over which the observed values are to be scaled. Whatever the set of numbers is, a location scale transform is applied to have the minimum value map to 0 and the maximum to 1.

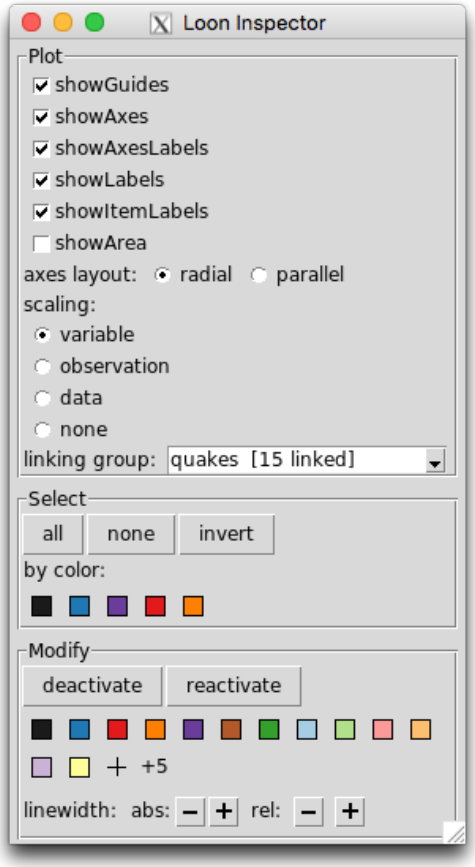
`scaling`	how the min and max are determined
`variable`	separate min and max found for each variate/axis/column (default)
`observation`	separate min and max found for each observation/row
`data`	min and max are determined using all values of all variates and all observations
`none`	no scaling is done, raw observation values are used

l_serialaxes() - radial coordinates

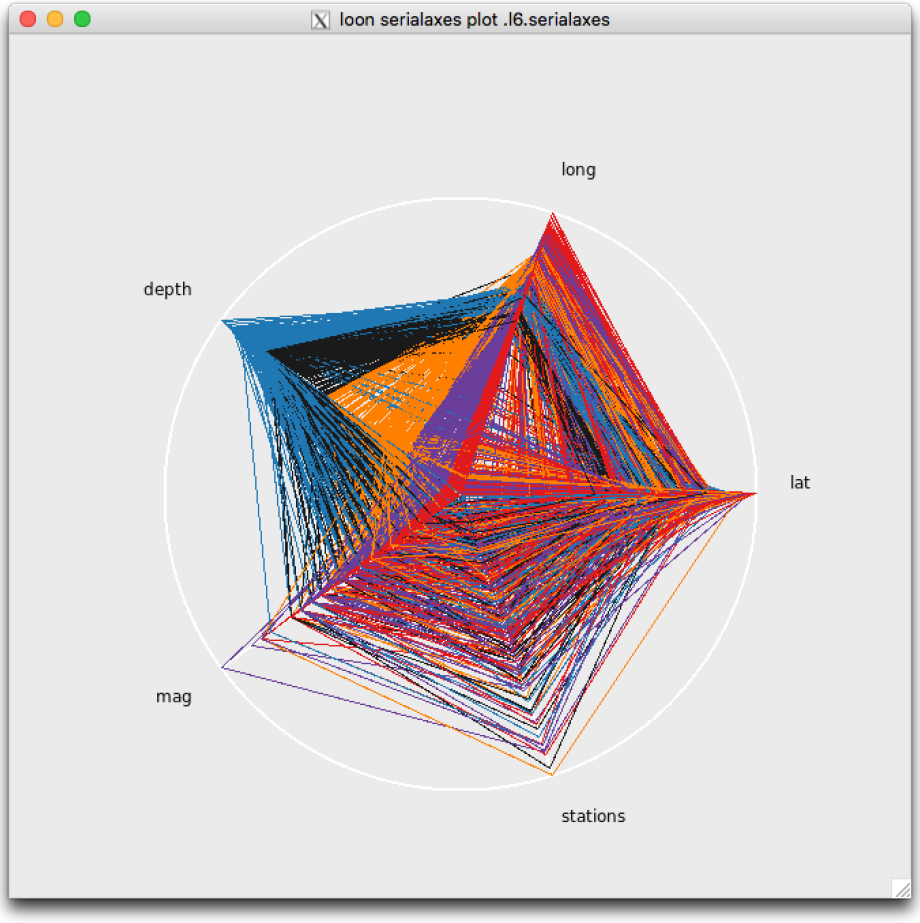
First reset the sequence to the original five and then change the layout of the axes:

```
serialAxes["sequence"] <- names(quakes)
serialAxes["axesLayout"] <- "radial"
```

Serial axes inspector



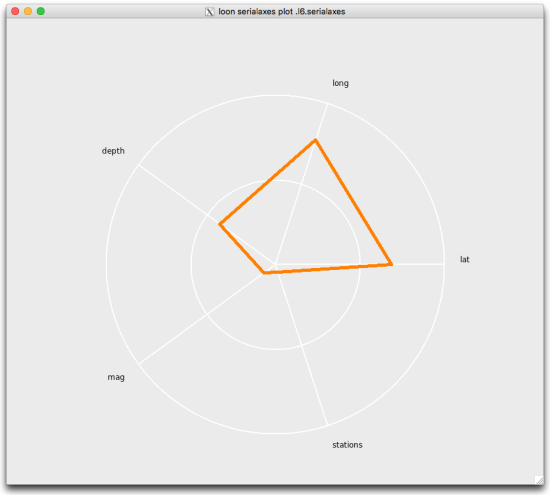
Radial (serial) axes plot

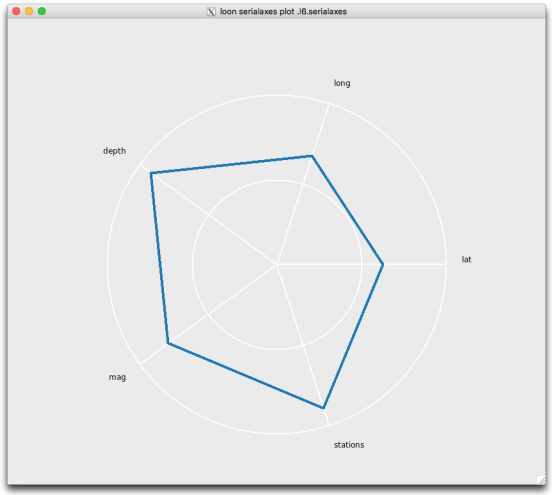


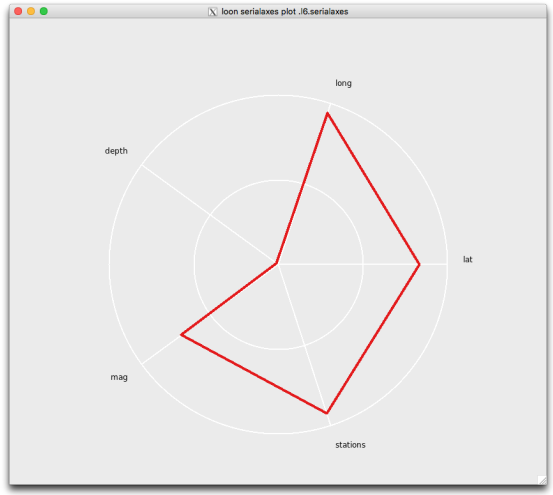
Each point in the 5-dimensional space becomes a "shape" in 2d.

l_serialaxes() - radial coordinates

For example, here are three different earth quakes:





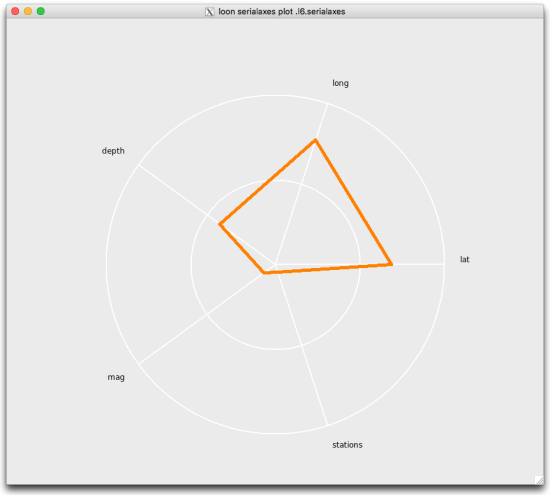


As can be seen, different observations will have different shapes, depending on their variate values. Those which have similar shape and size will be near each other in the (scaled) higher dimensional space. Similar shapes, but different sizes however do not indicate nearness in that space.
For example identical shapes of diminishing size will all lie along a line with the smallest shapes being nearer the origin (as defined by the minimum value on all variates).
The appearance of the glyph depends on the scaling used. The above use the default "variable" scaling.

`l_serialaxes()` - *radial coordinates*

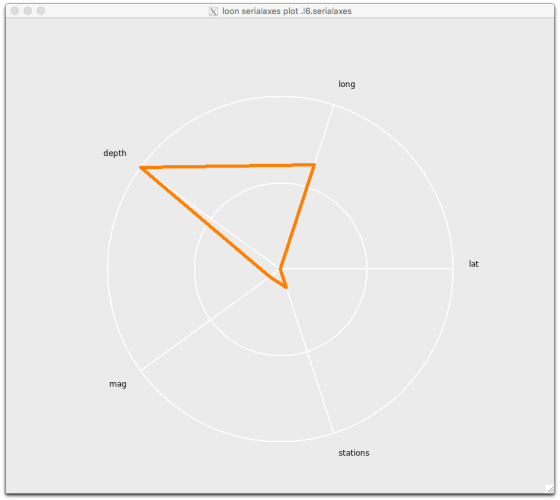
For example, here are three different scalings for the first of these quakes:

Variable scaling



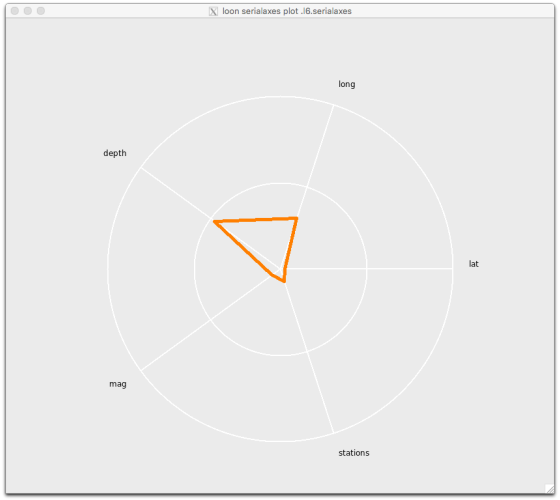
Each axis scaled to fit range of that variate.

Observation scaling



Axes scaled to observation's min and max across variates.

Data scaling



Identical scaling on all variates and observations.

Which scaling is important depends on the problem.

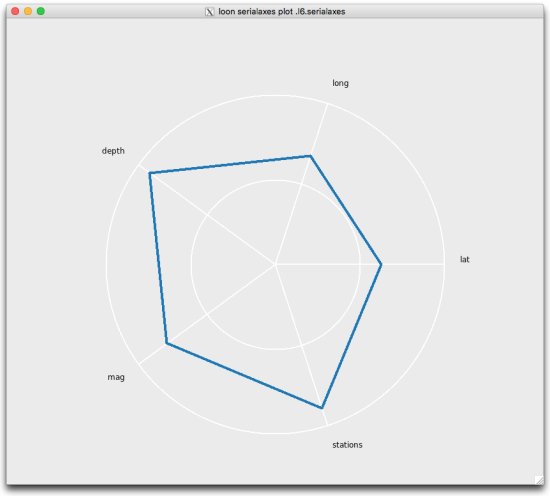
- Variable scaling is most common and generally applicable.
- Observation scaling is useful to compare observations on the basis of which variates they score highest/lowest on, without regard for how that score compares in magnitude to other observations.
- Data scaling is useful to compare observations by relative magnitude of all values. It is analogous to using Cartesian coordinates where all axes are on the same scale. No scaling ("none") is used when the data are all in $[0, 1]$ and we wish to preserve the actual values.

UNIVERSITY OF WATERLOO

`l_serialaxes()` - *radial coordinates*

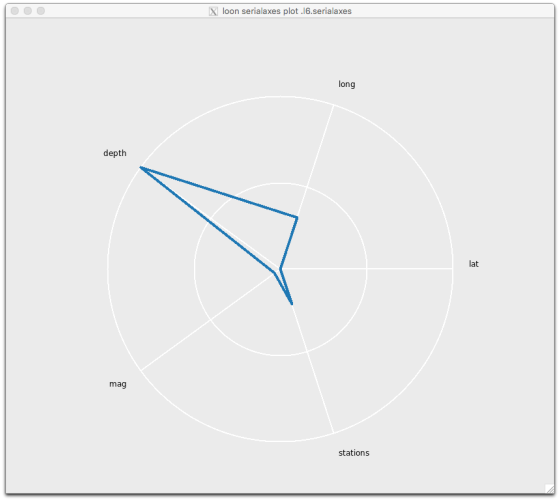
And the three different scalings for the second quake:

Variable scaling



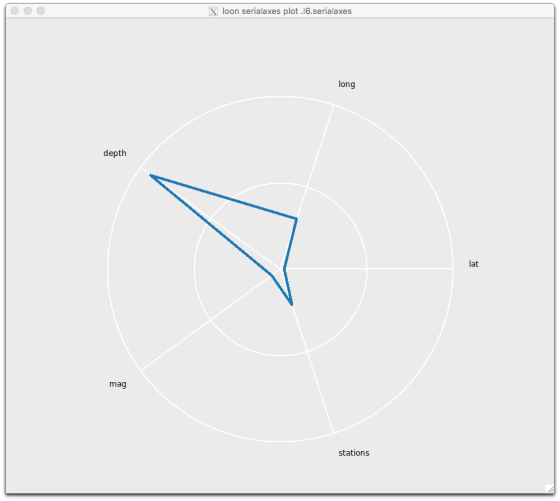
Each axis scaled to fit range of that variate.

Observation scaling



Axes scaled to observation's min and max across variates.

Data scaling



Identical scaling on all variates and observations.

Which scaling is important depends on the problem.

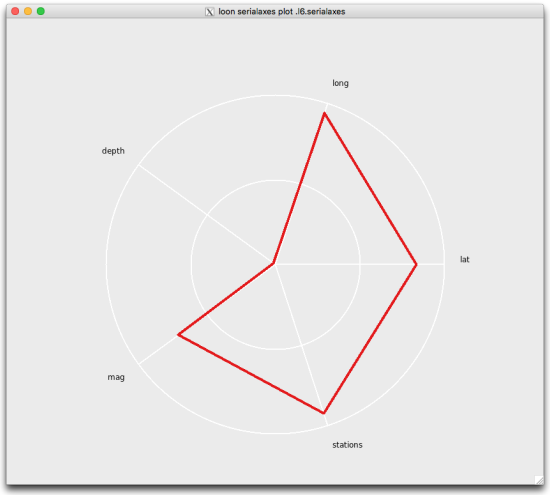
- Variable scaling is most common and generally applicable.
- Observation scaling is useful to compare observations on the basis of which variates they score highest/lowest on, without regard for how that score compares in magnitude to other observations.
- Data scaling is useful to compare observations by relative magnitude of all values. It is analogous to using Cartesian coordinates where all axes are on the same scale. No scaling ("none") is used when the data are all in $[0, 1]$ and we wish to preserve the actual values.

UNIVERSITY OF WATERLOO

`l_serialaxes()` - *radial coordinates*

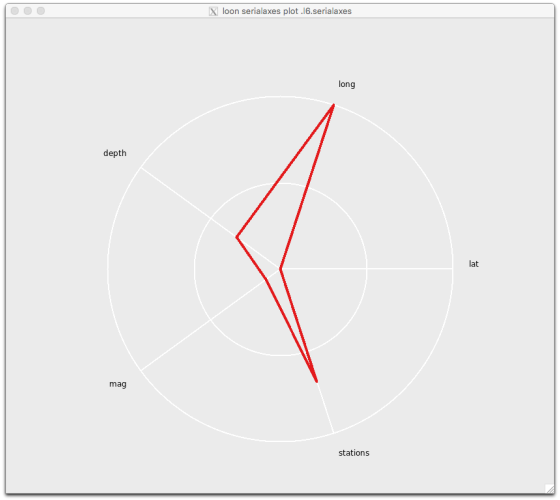
And finally the three different scalings for the third quake:

Variable scaling



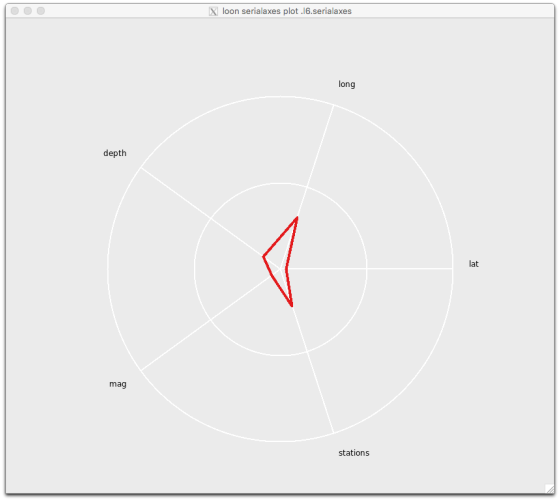
Each axis scaled to fit range of that variate.

Observation scaling



Axes scaled to observation's min and max across variates.

Data scaling



Identical scaling on all variates and observations.

Which scaling is important depends on the problem.

- Variable scaling is most common and generally applicable.
- Observation scaling is useful to compare observations on the basis of which variates they score highest/lowest on, without regard for how that score compares in magnitude to other observations.
- Data scaling is useful to compare observations by relative magnitude of all values. It is analogous to using Cartesian coordinates where all axes are on the same scale. No scaling ("none") is used when the data are all in $[0, 1]$ and we wish to preserve the actual values.

l_serialaxes() - radial coordinates

A more obvious value of the different scaling might be found in Statistics Canada's census data on visible minority populations in Canadian cities. These are available in loon as

```
data("minority")
str(minority)
```

```
## 'data.frame':    33 obs. of  18 variables:
##  $ Arab                                    : num  190 3840 165 125 195 ...
##  $ Black                                   : num  620 13270 1035 1250 330 ...
##  $ Chinese                                 : num  990 3100 295 975 295 ...
##  $ Filipino                                : num  155 530 100 205 50 ...
##  $ Japanese                                : num  65 410 10 10 0 170 30 0 2990 1800 ...
##  $ Korean                                  : num  45 620 65 120 15 ...
##  $ Latin.American                          : num  320 690 95 210 280 ...
##  $ Multiple.visible.minority               : num  25 780 175 60 10 ...
##  $ South.Asian                             : num  890 2900 350 485 45 ...
##  $ Southeast.Asian                         : num  55 655 65 60 55 ...
##  $ Total.population                        : num  179270 369455 124055 120875 149600 ...
##  $ Visible.minority.not.included.elsewhere: num  40 180 10 30 10 ...
##  $ Visible.minority.population             : num  3460 27645 2425 3805 1280 ...
##  $ West.Asian                              : num  65 670 70 270 0 ...
##  $ lat                                     : num  47.6 44.7 46.1 45.3 48.4 ...
##  $ long                                    : num  -52.7 -63.6 -64.8 -66.1 -71.1 ...
##  $ googleLat                               : num  47.6 44.6 46.1 45.3 48.4 ...
##  $ googleLong                              : num  -52.7 -63.6 -64.8 -66.1 -71.1 ...
```

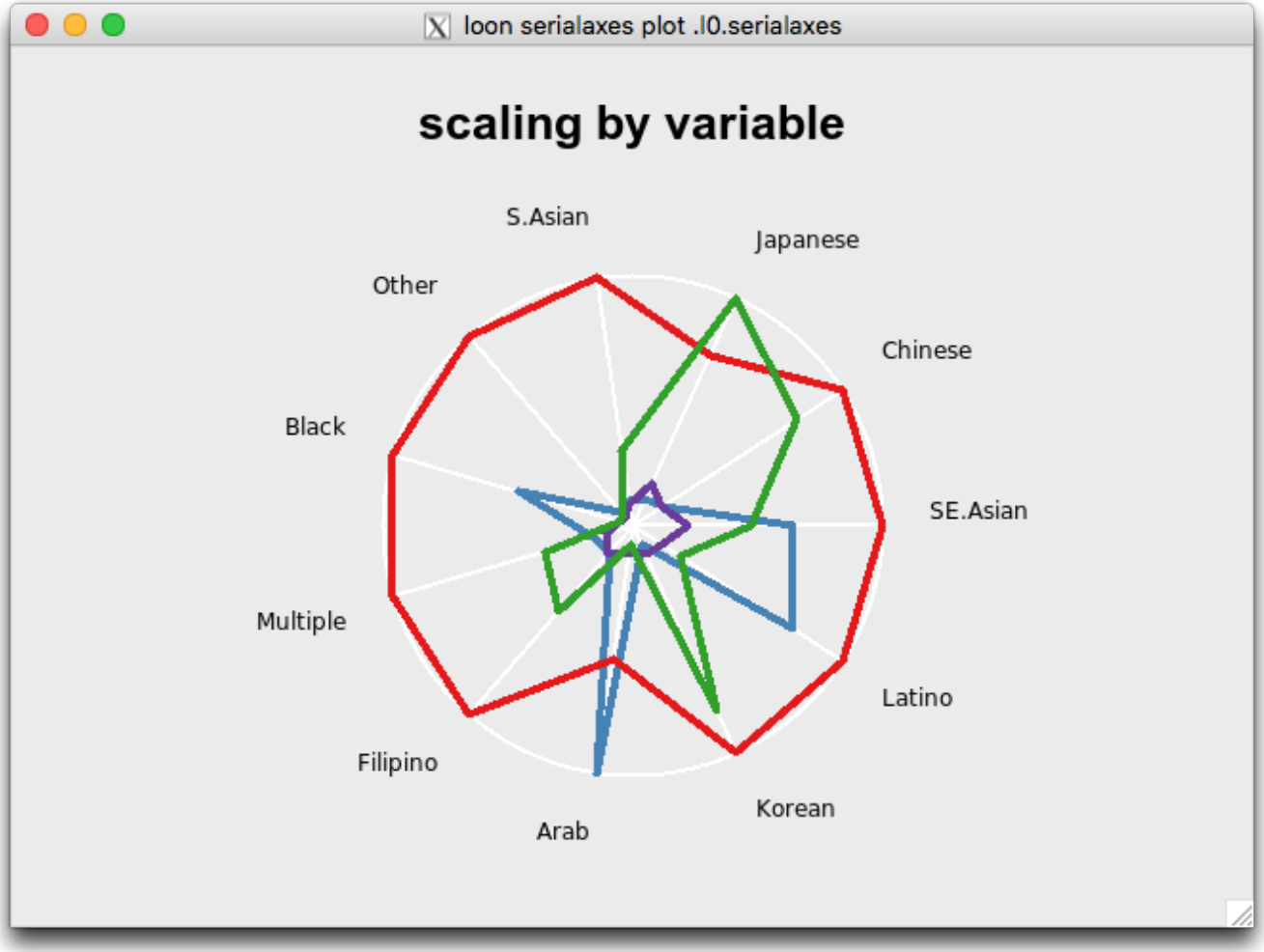
These contain the population of various self-declared "visible minorities" in 33 cities across Canada.

UNIVERSITY OF WATERLOO

l_serialaxes() - radial coordinates

Effect of scaling on visible minority data in four large Canadian cities (west to east): Vancouver (green), Calgary (purple), Toronto (red), and Montreal (blue):

Variable scaling

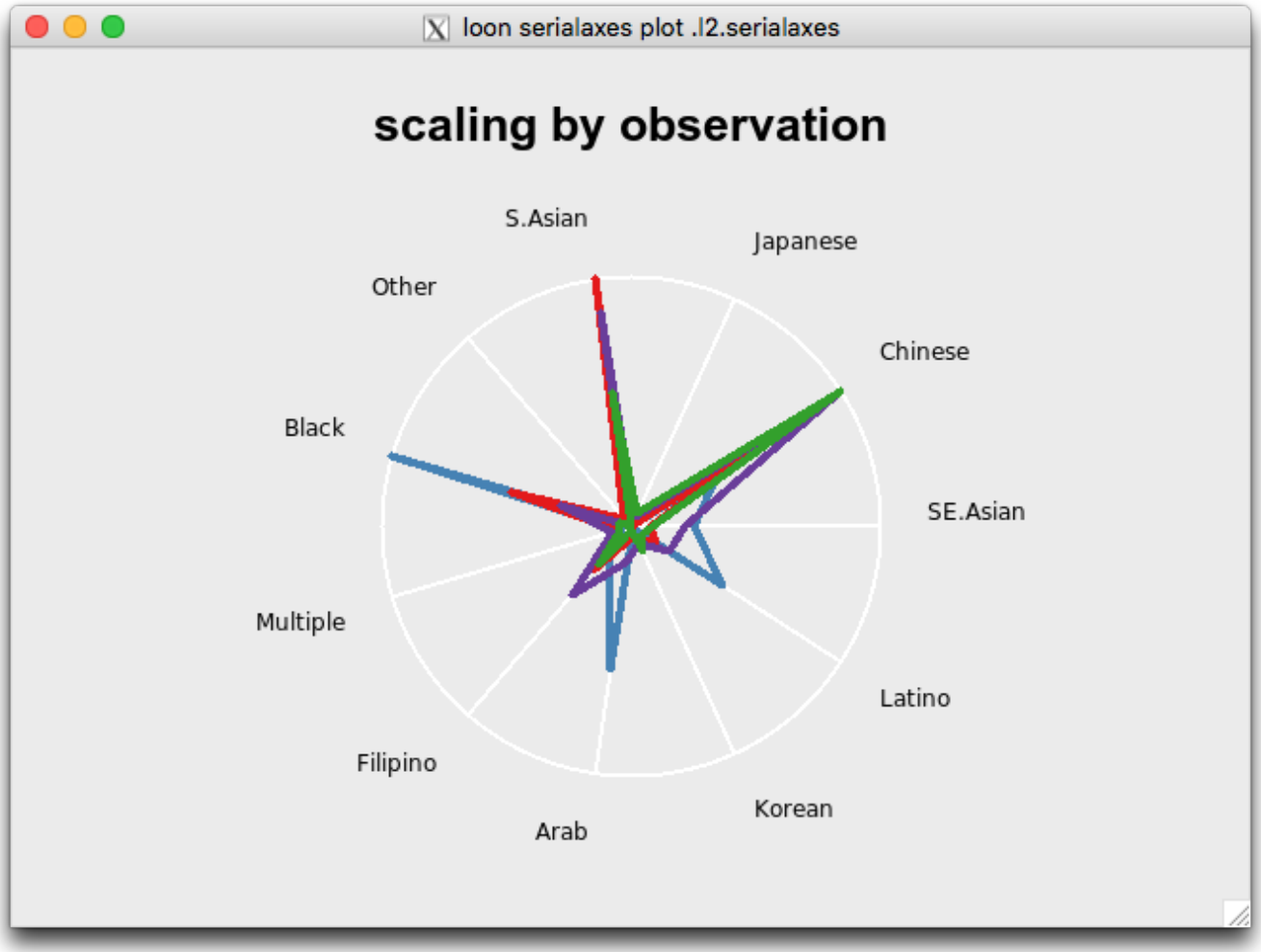


Each axis scaled to fit range of that variate.

l_serialaxes() - radial coordinates

Effect of scaling on visible minority data in four large Canadian cities (west to east): Vancouver (green), Calgary (purple), Toronto (red), and Montreal (blue):

Observation scaling

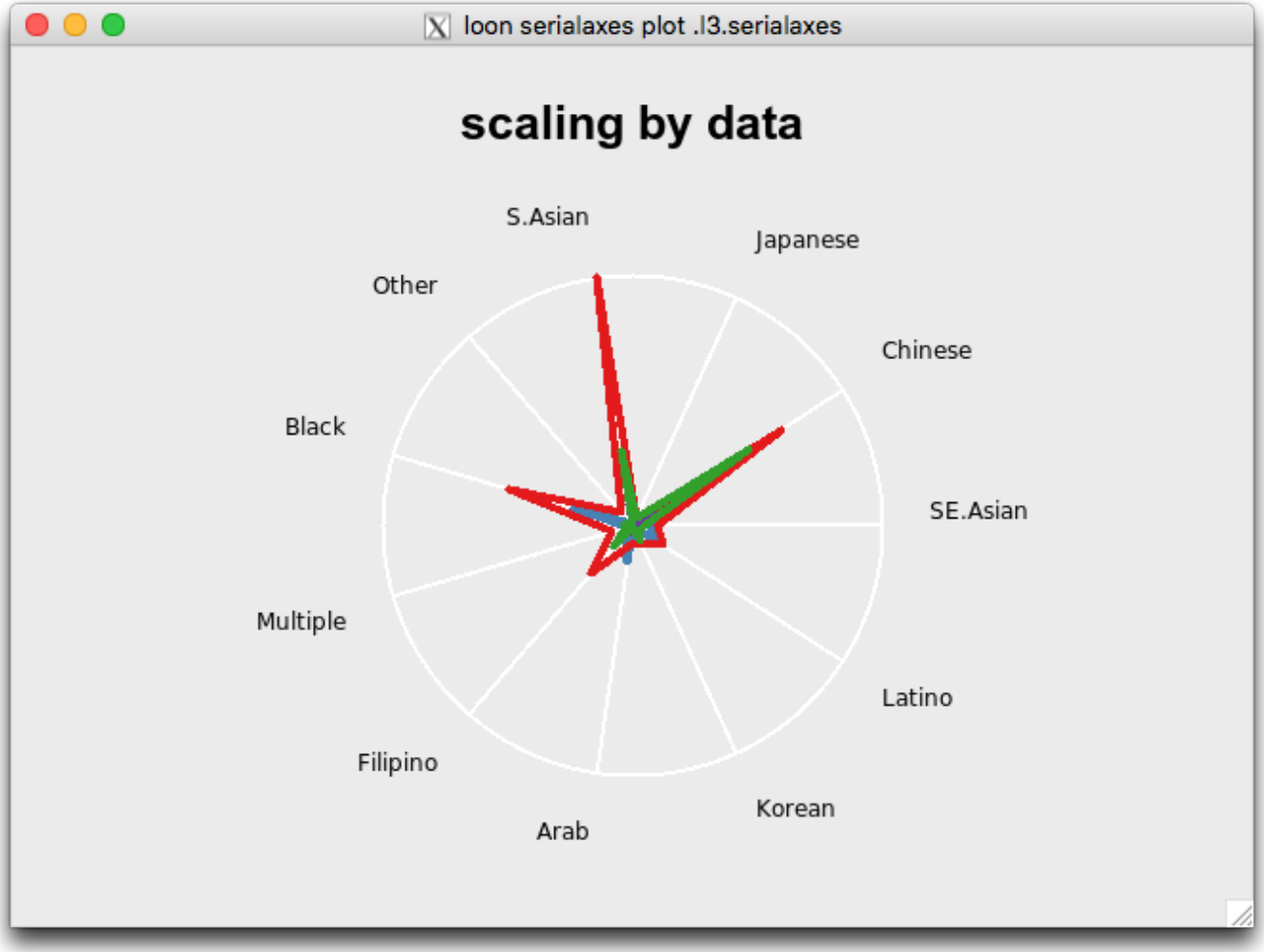


Axes scaled to observation's min and max across variates.

l_serialaxes() - radial coordinates

Effect of scaling on visible minority data in four large Canadian cities (west to east): Vancouver (green), Calgary (purple), Toronto (red), and Montreal (blue):

Data scaling

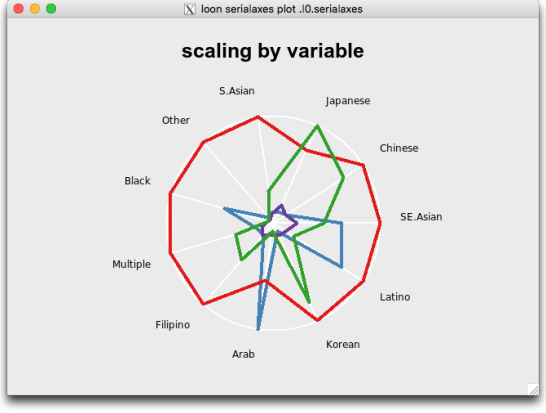


Identical scaling on all variates and observations.

`l_serialaxes()` - *radial coordinates*

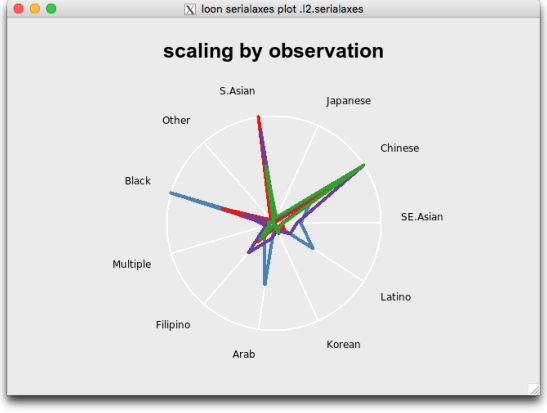
Effect of scaling on visible minority data in four large Canadian cities (west to east): Vancouver (green), Calgary (purple), Toronto (red), and Montreal (blue):

Variable scaling



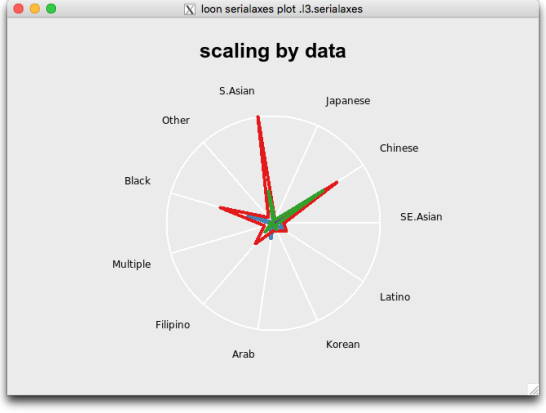
Each axis scaled to fit range of that variate.

Observation scaling



Axes scaled to observation's min and max across variates.

Data scaling



Identical scaling on all variates and observations.

UNIVERSITY OF WATERLOO

Serial axes as glyphs - `l_glyph_add_serialaxes()`

Because serial axes can show many dimensions at once, they are also useful as glyphs in a scatterplot.

For example, a radial axis plot will produce a unique glyph for each observation. Similarities aband differences between glyph appearances will provide some information on all variates.

Unfortunately, these glyphs cannot participate in linking between plots. They have much internal structure which we might want to change separately for each plot and this is the main reason that "glyph" is not one of the linked states by default.

So we remove `"glyph"` from the linked states of the plots to get back to the default behaviour (NB. only because we previously added it to the linked states).

```
for (plot in plots) {
  l_setLinkedStates(plot,
                    setdiff(l_getLinkedStates(plot),
                            "glyph"))
}
```

Now serial axes glyphs can be attached to any of these plots without causing any conflict due to linking.

Serial axes as glyphs - `l_glyph_add_serialaxes()`

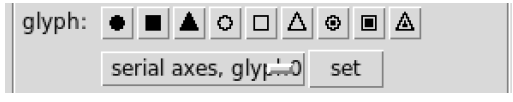
Serial axes glyphs must be created and added to the plot structure before being displayed.

For example, suppose that we were to add serial axes glyphs to the first plot `p`. This is accomplished in two steps.

First the glyph is constructed and added to the plot structure:

```
# Construction and attachment of serial axes as glyphs
serAxesGlyphs <-  l_glyph_add_serialaxes(p, data = quakes,
                                         label = "serial axes")
```

Second, selected points can now have their glyph be set to a serial axes glyph. Typically this is done via its loon inspector. Just as with the default shapes, simply select the points to be modified, select the glyph (from a menu if more than one such glyph has been constructed) and click on set:



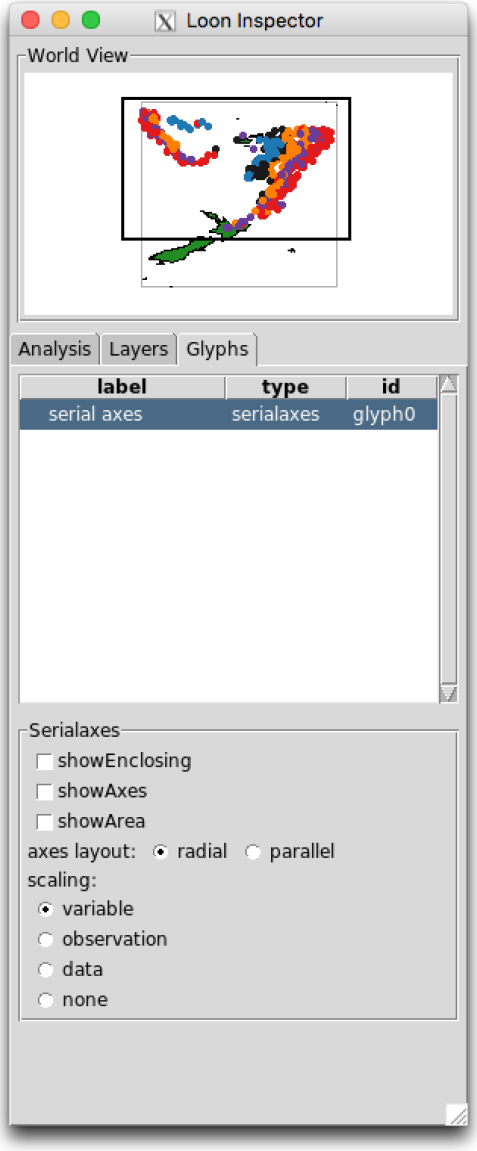
If, as above, the value from `l_glyph_add_serialaxes()` has been saved, then this value may be used to programmatically set the glyph for any subset of points as

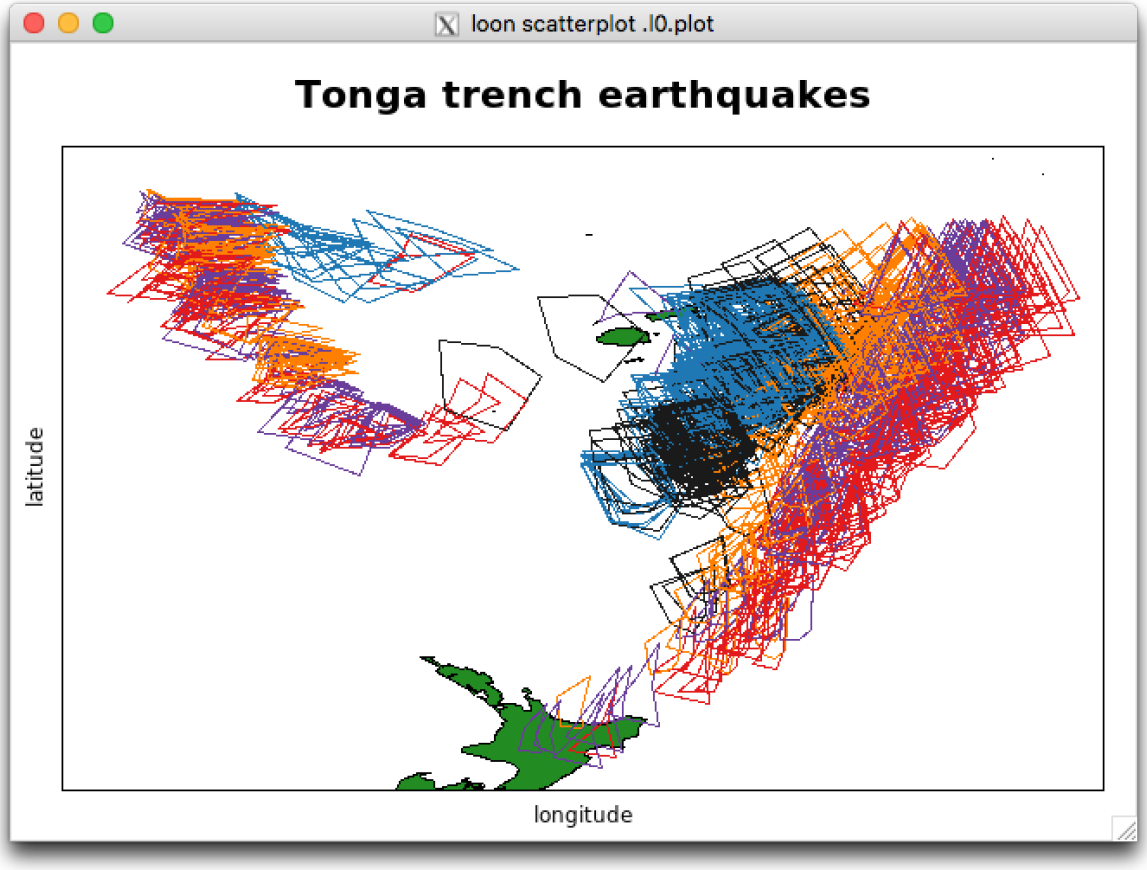
```
p['glyph'] <- serAxesGlyphs
```

Either way, the glyphs of the points in `p` can now appear in its display.

Serial axes as glyphs

As glyphs, serial axes have structure that is accessible via the "Glyphs" tab on the loon inspector.





Now every observation is displayed (with linked colour, etc.) as a serial axes glyph.

Serial axes as glyphs

The same functionality, and a little more, can be accessed programmatically:

```
names(serAxesGlyphs)
```

```
##  [1] "showAxes"       "showAxesLabels" "linewidth"      "scaling"
##  [5] "axesLayout"     "showArea"       "axesLabels"     "data"
##  [9] "sequence"       "showEnclosing"  "bboxColor"      "axesColor"
```

```
serAxesGlyphs["axesLabels"]
```

```
## [1] "lat"      "long"     "depth"    "mag"      "stations"
```

The states "showAxesLabels", "linewidth", "axesLabels", "data", "sequence", "bboxColor", and "axesColor" can only be changed programmatically (provided the value has been saved beforehand as was done with `serAxesGlyphs`). Of these, only those associated with the display, namely "linewidth", "bboxColor", and "axesColor", should be changed programmatically. The others should be determined only at the time of construction.

```
serAxesGlyphs["linewidth"] <- 2
serAxesGlyphs["bboxColor"] <- "black"
serAxesGlyphs["axesColor"] <- "red"
```

Note that one of the states is "data" which suggests that different data could be used to create the glyphs than that which created the plot. As long as the number of observations is the same, this would work and could in some instances be very useful.

Serial axes as glyphs - `l_glyph_add_serialaxes()`

We might have a longer sequence and make this available on all plots (including p):

```
# Use a sequence to have all pairs appear
seqVars <- names(quakes)[c(1, 2, 3, 1, 4, 2, 5, 3, 4, 5)]

for (plot in plots) {
    # Add the serial axes as glyphs to scatterplots
    if (is(plot, "l_plot")) {
        l_glyph_add_serialaxes(plot,
                               axesLayout = "radial",
                               data = quakes,
                               sequence = seqVars,
                               scaling = "variable",
                               showArea=TRUE,
                               label = "serial axes: all pairs")
    }
}
```

The plots can now be used in conjunction and various areas explored for like shapes and/or unusual shapes.

The "minority" vignette provides another example where the serial axes glyphs are more easily interpreted.

Programmatic access when a loon plot was not assigned at creation

Sometimes a loon plot is created without assigning it to a variable. This can be a nuisance whenever we later want to interact with the plot programmatically.

For example, the plot would have been earlier created (unassigned) as

```
l_plot(quakes[, "long"], quakes[,"lat"]) # which would have printed as
```

```
## [1] ".l8.plot"
## attr(,"class")
## [1] "l_plot" "loon"
```

The printed value has all the necessary information. It consists of the plot's `tcltk` name as a string (notice the leading dot followed by the lower case letter L) and a vector of the `loon` class inheritance. Note that the name string also appears appended to the title of the plot's window.

Even after its creation, *full programmatic interaction can still be made available to any `loon` plot via a variable*, say `myplot`, *in two simple steps*:

1. Assign the `tcltk` name string as the value of a variable as in
   `myplot <-` ".l8.plot".
2. Assign the correct class hierarchy to variable, here `myplot`
   `class(myplot) <- c("l_plot", "loon")`

Now the variable `myplot` can be used just **as if the `loon` plot had been assigned to it at creation**.

NOTE It is enough to just assign the `"loon"` class, as in `class(myplot) <- "loon"`, for access `myplot[]` and setting `myplot[] <-` to work for names which would appear in `names(myplot)`. However, this is **not** generally recommended because other important functionality like `plot(myplot)` will not work without the full class hierarchy. Fortunately, the command history and output is often accessible and contains the original output at creation; alternatively, constructing any other plot with the same command will also reveal the appropriate class hierarchy needed for full `loon` functionality and the `tcltk` name usually appears in the window title.

UNIVERSITY OF WATERLOO

Fun exploration 1: Human immunoglobulin G1 antibody molecule

Here, you will explore the three dimensional structure of the (alpha) carbon atoms making up a single molecule.

```
library(loon.data)
data(igg1)
names(igg1)
```

```
##  [1] "recordType"         "name"               "residue"
##  [4] "chainID"            "residueSequenceNum" "x"
##  [7] "y"                  "z"                  "residueName"
## [10] "group"
```

Fun exploration 1: Human immunoglobulin G1 antibody molecule

Here, you will explore the three dimensional structure of the (alpha) carbon atoms making up a single molecule.

```
library(loon.data)
data(igg1)
names(igg1)
```

```
##  [1] "recordType"         "name"               "residue"
##  [4] "chainID"            "residueSequenceNum" "x"
##  [7] "y"                  "z"                  "residueName"
## [10] "group"
```

Now explore the molecule shape as it depends on other variables.

For example, start with

```
igg1_3d <- with(igg1,
                l_plot3D(x, y, z,
                         linkingGroup = "igg1",
                         itemLabel =
                             paste0("Residue: ", residueName, " (",
                                    residue, ")","\n",
                                   "Group: ", group),
                         showItemLabels = TRUE)
                )
```

See what you can learn about this molecule.

Fun exploration 2: 1974 Motor Trend Car Road Tests

"The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models)."

```
names(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

Fun exploration 2: 1974 Motor Trend Car Road Tests

"The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models)."

```
names(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
pp_mtcars1 <- l_pairs(mtcars[, c("mpg", "disp", "hp",
                                 "drat", "wt", "qsec")],
                      linkingGroup = "mtcars",
                      showHistograms = TRUE,
                      itemLabel = row.names(mtcars),
                      showItemLabels = TRUE)

pp_mtcars2 <- l_pairs(mtcars[, c("cyl", "vs", "am",
                                 "gear", "carb")],
                      linkingGroup = "mtcars",
                      showHistograms = TRUE,
                      itemLabel = row.names(mtcars),
                      showItemLabels = TRUE)
fit <- lm(mpg ~ . , data = mtcars)
summary(fit)
```